

Gilles Grimaud
François-Xavier Standaert (Eds.)

LNCS 5189

Smart Card Research and Advanced Applications

8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008
London, UK, September 2008
Proceedings



ifip

 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Gilles Grimaud
François-Xavier Standaert (Eds.)

Smart Card Research and Advanced Applications

8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008
London, UK, September 8-11, 2008
Proceedings

Volume Editors

Gilles Grimaud
IRCICA/LIFL, CNRS UMR 8022
Univ. Lille 1, INRIA Lille - Nord Europe
Université des Sciences et Technologies de Lille LIFL
Batiment M3, 59655 cité scientifique, France
E-mail: gilles.grimaud@inria.fr

François-Xavier Standaert
UCL Crypto Group
Microelectronics Laboratory
Place du Levant, 3, 1348 Louvain-la-Neuve, Belgium
E-mail: fstandae@uclouvain.be

Library of Congress Control Number: 2008933705

CR Subject Classification (1998): E.3, K.6.5, C.3, D.4.6, K.4.1, E.4, C.2

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743
ISBN-10 3-540-85892-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85892-8 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© IFIP International Federation for Information Processing, Hofstrasse 3, A-2361 Laxenburg, Austria 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12513536 06/3180 5 4 3 2 1 0

Preface

Since 1994, CARDIS has been the foremost international conference dedicated to smart card research and applications. Every two years, the scientific community congregates to present new ideas and discuss recent developments with both an academic and industrial focus. Following the increased capabilities of smart cards and devices, CARDIS has become a major event for the discussion of the various issues related to the use of small electronic tokens in the process of human-machine interactions. The scope of the conference includes numerous subfields such as networking, efficient implementations, physical security, biometrics, and so on.

This year's CARDIS was held in London, UK, on September 8–11, 2008. It was organized by the Smart Card Centre, Information Security Group of the Royal Holloway, University of London.

The present volume contains the 21 papers that were selected from the 51 submissions to the conference. The 22 members of the program committee worked hard in order to evaluate each submission with at least three reviews and agree on a high quality final program. Additionally, 61 external reviewers helped the committee with their expertise. Two invited talks completed the technical program. The first one, given by Ram Banerjee and Anki Nelaturu, was entitled “Getting Started with Java Card 3.0 Platform”. The second one, given by Aline Gouget, was about “Recent Advances in Electronic Cash Design” and was completed by an abstract provided in these proceedings.

We would like to express our deepest gratitude to the various people who helped in the organization of the conference and made it a successful event. In the first place, we thank the authors who submitted their work and the reviewers who volunteered to discuss the submitted papers over several months. We also acknowledge the work of our invited speakers. The assistance of the Smart Card Centre at Royal Holloway was greatly appreciated. We are particularly grateful to Konstantinos Markantonakis and Keith Mayes, the organizing committee co-chairs. A big thank-you to Damien Sauveron, who maintained the submission webtool, and to the staff at Springer for solving the practical publication issues. And finally, we would like to thank the CARDIS steering committee for allowing us to serve at such a recognized conference.

September 2008

Gilles Grimaud
François-Xavier Standaert

**Smart Card Research and Advanced
Applications
8th IFIP WG 8.8/11.2 International Conference
CARDIS 2008**

London, UK, September 2008

Organizing Committee

Konstantinos Markantonakis Royal Holloway, University of London, UK
Keith Mayes Royal Holloway, University of London, UK

Program Committee

Mehdi-Laurent Akkar	Barclays Capital, USA
Gildas Avoine	Université catholique de Louvain, Belgium
Boris Balacheff	Hewlett-Packard Laboratories, UK
Eduardo De Jong	Sun Microsystems, USA
Josep Domingo-Ferrer	Universitat Rovira i Virgili, Spain
Dieter Gollmann	TU Hamburg-Harburg, Germany
Louis Goubin	Université de Versailles, France
Gilles Grimaud	University of Lille 1, France (co-chair)
Pieter Hartel	University of Twente, The Netherlands
Jaap-Henk Hoepman	Radbout University Nijmegen, The Netherlands
Dirk Husemann	IBM Zurich Research Laboratories, Switzerland
Marc Joye	Thomson Multimedia, France
Jean-Louis Lanet	GemAlto, France
Javier Lopez	University of Malaga, Spain
Pierre Paradinas	INRIA, France
Joachim Posegga	University of Hamburg, Germany
Emmanuel Prouff	Oberthur Card Systems, France
Damien Sauveron	University of Limoges, France
Isabelle Simplot-Ryl	University of Lille, France
François-Xavier Standaert	UCL Crypto Group, Belgium (co-chair)
Issa Taore	University of Victoria, Canada
Mike Tunstall	University College Cork, Ireland
Jean-Jacques Vandewalle	GemAlto, France
Johannes Wolkerstorfer	IAIK/ University of Graz, Austria

External Reviewers

Antoni Martinez Balleste	Alain Durand	Robert Naciri
Claude Barral	Pierre Dusart	Gilles Piret
Salvatore Bocchetti	Martin Feldhofer	Henrich C. Poehls
Pierre-François Bonnefoi	Pierre Girard	Emanuel Popovici
Arnaud Boscher	Sylvain Guilley	Thomas Popp
Samia Bouzefrane	Stuart Haber	Christian Rechberger
Bastian Braun	Georg Hofferek	Mathieu Rivain
Emmanuel Bresson	Michael Hutter	Tomas Sander
Ileana Buhan	Samuel Hym	Daniel Schreckling
Jordi Castella-Roca	Luan Ibraimi	Francesc Sebe
Serge Chaumette	François Ingelrest	Saeed Sedghi
Liqun Chen	Martin Johns	Yannick Sierra
Christophe Clavier	Chong Hee Kim	Sergei Skorobogatov
Julien Cordry	Markus Kuhn	Agusti Solanas
Mark Crosbie	Cedric Lauradoux	Junko Takahashi
Vanesa Daza	François Mace	Jean-Marc Talbot
Lauren Del Giudice	Mark Manulis	Ronald Toegl
Eric Deschamps	Nathalie Mitton	Claire Whelan
Trajce Dimkow	Ayse Morali	Emmanuele Zambon
Roberto Di Pietro	Christophe Mourtel	
Emmanuelle Dottax	Christophe Muller	

Table of Contents

Malicious Code on Java Card Smartcards: Attacks and Countermeasures	1
<i>Wojciech Mostowski and Erik Poll</i>	
Static Program Analysis for Java Card Applets	17
<i>Vasilios Almaliotis, Alexandros Loizidis, Panagiotis Katsaros, Panagiotis Louridas, and Diomidis Spinellis</i>	
On Practical Information Flow Policies for Java-Enabled Multiapplication Smart Cards	32
<i>Dorina Ghindici and Isabelle Simplot-Ryl</i>	
New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough	48
<i>Chong Hee Kim and Jean-Jacques Quisquater</i>	
DSA Signature Scheme Immune to the Fault Cryptanalysis	61
<i>Maciej Nikodem</i>	
A Black Hen Lays White Eggs: Bipartite Multiplier Out of Montgomery One for On-Line RSA Verification	74
<i>Masayuki Yoshino, Katsuyuki Okeya, and Camille Vuillaume</i>	
Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents	89
<i>Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar</i>	
Fast Hash-Based Signatures on Constrained Devices	104
<i>Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar</i>	
Fraud Detection and Prevention in Smart Card Based Environments Using Artificial Intelligence	118
<i>Wael William Zakhari Malek, Keith Mayes, and Kostas Markantonakis</i>	
The Trusted Execution Module: Commodity General-Purpose Trusted Computing	133
<i>Victor Costan, Luis F.G. Sarmenta, Marten van Dijk, and Srinivas Devadas</i>	
Management of Multiple Cards in NFC-Devices	149
<i>Gerald Madlmayr, Oliver Dillinger, Josef Langer, and Josef Scharinger</i>	

Coupon Recalculation for the GPS Authentication Scheme	162
<i>Georg Hofferek and Johannes Wolkerstorfer</i>	
Provably Secure Grouping-Proofs for RFID Tags	176
<i>Mike Burmester, Breno de Medeiros, and Rossana Motta</i>	
Secure Implementation of the Stern Authentication and Signature Schemes for Low-Resource Devices	191
<i>Pierre-Louis Cayrel, Philippe Gaborit, and Emmanuel Prouff</i>	
A Practical DPA Countermeasure with BDD Architecture	206
<i>Toru Akishita, Masanobu Katagi, Yoshikazu Miyato, Asami Mizuno, and Kyoji Shibutani</i>	
SCARE of an Unknown Hardware Feistel Implementation	218
<i>Denis Réal, Vivien Dubois, Anne-Marie Guilloux, Frédéric Valette, and Mhamed Drissi</i>	
Evaluation of Java Card Performance	228
<i>Samia Bouzefrane, Julien Cordry, Hervé Meunier, and Pierre Paradinas</i>	
Application of Network Smart Cards to Citizens Identification Systems	241
<i>Joaquin Torres, Mildrey Carbonell, Jesus Tellez, and Jose M. Sierra</i>	
SmartPRO: A Smart Card Based Digital Content Protection for Professional Workflow	255
<i>Alain Durand, Marc Éluard, Sylvain Lelievre, and Christophe Vincent</i>	
A Practical Attack on the MIFARE Classic	267
<i>Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia</i>	
A Chemical Memory Snapshot	283
<i>Jörn-Marc Schmidt</i>	
Recent Advances in Electronic Cash Design	290
<i>Aline Gouget</i>	
Author Index	295

Malicious Code on Java Card Smartcards: Attacks and Countermeasures

Wojciech Mostowski and Erik Poll

Digital Security (DS) group, Department of Computing Science
Radboud University Nijmegen, The Netherlands
{woj,erikpoll}@cs.ru.nl

Abstract. When it comes to security, an interesting difference between Java Card and regular Java is the absence of an on-card bytecode verifier on most Java Cards. In principle this opens up the possibility of malicious, ill-typed code as an avenue of attack, though the Java Card platform offers some protection against this, notably by code signing.

This paper gives an extensive overview of vulnerabilities and possible runtime countermeasures against ill-typed code, and describes results of experiments with attacking actual Java Cards currently on the market with malicious code.

1 Overview

A huge security advantage of type safe language such as Java is that the low level memory vulnerabilities, which plague C/C++ code in the form of buffer overflows, are in principle ruled out. Also, it allows us to make guarantees about the behaviour of one piece of code, without reviewing or even knowing all the other pieces of code that may be running on the same machine.

However, on Java Card smartcards [\[9\]](#) an on-card bytecode verifier (BCV) is only optional, and indeed most cards do not include one. This means that malicious, ill-typed code is a possible avenue of attack.

Of course, the Java Card platform offers measures to protect against this, most notably by restricting installation of applets by means of digital signatures – or disabling it completely. Still, even if most Java Card smartcards that are deployed rely on these measures to avoid malicious code, it remains an interesting question how vulnerable Java Card applications are to malicious code. Firstly, the question is highly relevant for security evaluations of code: can we evaluate the code of one applet without looking at other applets that are on the card? Secondly, the defence mechanisms of the Java Card platform are not so easy to understand; for instance, there is the firewall as an extra line of defence, but does that offer any additional protection against ill-typed code, and can it compensate for the lack of BCV? And given the choice between cards with and without BCV, are there good reasons to choose for one over the other? (As we will show, cards with on-card BCV may still be vulnerable to ill-typed code!)

In this paper we take a serious look at the vulnerability of the Java Card platform against malicious, ill-typed code. We consider the various ways to get

ill-typed code on a smartcard, and various ways in which one applet could try to do damage to another applet or the platform, and the countermeasures the platform might deploy.

There is surprisingly little literature on these topics. The various defences of the Java Card platform are only given rather superficial discussion in [6, Chapter 8]. The only paper we know that discusses type flaw attacks on a Java Card smartcard is [11].

We have experimented with attacks on eight different cards from four manufacturers, implementing Java Card versions 2.1.1, 2.2, or 2.2.1. We will refer to these cards as A_211, A_221, B_211, B_22, B_221, C_211A, C_211B, and D_211. The first letter indicates the manufacturer, the numbers indicate which Java Card version the card provides. Based on the outcome of the experiments, we can make some educated guesses on which of the countermeasures the cards actually implement.

The outline of the rest of this paper is as follows. Sect. 2 briefly reviews the different lines of defence on the Java Card platform, including bytecode verification and the applet firewall. The first step in any attack involving ill-typed code is getting ill-typed code installed on the smartcard. More precisely, we want to somehow create type confusion, or break the type system, by having two pieces of code treat (a reference to) the *same* piece of physical memory as having different, incompatible types. Sect. 3 discusses the various ways to create type confusion and the success we had with these methods on the various cards. Sect. 4 then discusses experiments with concrete attacks scenarios.

Sect. 5 discusses the various runtime countermeasures the platform could implement against such attacks, some of which we ran into in the course of our experiments. Finally, Sect. 6 summarises our results and discusses the implications.

2 Defences

In this section we briefly describe and compare the protection provided by the various protection mechanisms on a Java Card platform.

2.1 Bytecode Verification

Bytecode verification of Java (Card) code guarantees type correctness of code, which in turn guarantees memory safety. For the normal Java platform, code is subjected to bytecode verification at load time. For Java Cards, which do not support dynamic class loading, bytecode verification can be performed at installation time (when an applet is installed on the smartcard). However, most Java Card smartcards do not have an on-card BCV, and check a digital signature of a third party who is trusted to have performed bytecode verification off-card.

Note that even if bytecode is statically verified, some checks will always have to be done dynamically, namely checks for non-nullness, array bounds, and downcasts. For Java Card, the applet firewall will also require runtime checks. Although typically bytecode verification is done statically, it is also possible to

do type checking dynamically, i.e. at runtime, by a so-called defensive virtual machine. This requires keeping track of typing information at runtime and performing type checks prior to the execution of every bytecode instruction. Clearly, this has an overhead both in execution time and in memory usage. However, to check downcasts the VM already has to record runtime types of objects anyway.

As we will see later, our experiments show that some Java Cards do a form of runtime type checking, and this then offers an excellent protection against ill-typed code.

2.2 Applet Firewall

The applet firewall is an additional defence mechanism implemented on all Java Cards. The firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets (of applets in a different security context, to be precise). For every object its context is recorded, and for any field or method access it is checked if it is allowed. In a nutshell, applets are only allowed to access in their own context, but the Java Card Runtime Environment (JCRE) has the right to access anything. In UNIX terminology, the JCRE executes in root-mode, and some of the Java Card API calls, which are executed in JCRE context, are ‘setuid root’.

Defence mechanisms can be *complementary*, each providing different guarantees that the other cannot, or *defence in depth*, each providing the same guarantees, so that one can provide a back-up in case the other fails. As defence mechanisms, the firewall and bytecode verification are primarily complementary. The firewall provides additional guarantees that bytecode verification does not: a carelessly coded applet might expose some of its data fields by declaring them public, allowing other applets to read or modify them.¹ Bytecode verification cannot protect against this, but the firewall will. The firewall provides a strong guarantee that an applet cannot be influenced by the behaviour of other (well-typed!) applets, unless it explicitly exposes functionality via a so-called Shareable Interface Object.

The firewall is not guaranteed to offer protection against ill-typed applets. Still, for certain attack scenarios, the firewall does provide a useful second line of defence. If a malicious applet manages to get hold of a ‘foreign’ reference to an object belonging to another applet by breaking the type system, its access to that object reference is still subject to runtime firewall checks, which may prevent any harm.

Breaking the Firewall. The firewall as specified in [9] is quite complicated. This means that there is a real chance of implementation bugs, or unclarities in the specs, which might lead to security flaws. We investigated the firewall specification and thoroughly tested cards for any flaws. Details are described in a separate technical report [8]. We did find some minor deviations from the

¹ Indeed, security coding guidelines for Java such as [6, Chapter 7] or <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/java.html> warn against the use of public, package, and protected (!) visibility.

official specification on some cards, but most of them are ‘safe’, in the sense that the cards were more restrictive than the specification demanded. The only ‘unsafe’ violation of the specifications we found was that card `A_221` ignores the restriction that access via a shareable interface should not be allowed when an applet is active on another logical channel. This could lead to security problems in particular applications that use shareable interfaces. The tests in [8] only consider well-typed code. However, a weak firewall implementation can be broken with ill-typed code, as we will discuss in Sect. 4.2.4.

3 Getting Ill-Typed Code on Cards

We know four ways to get ill-typed code onto a smartcard: (i) CAP file manipulation, (ii) abusing shareable interface objects, (iii) abusing the transaction mechanism, and (iv) fault injection. Note that an on-card BCV is only guaranteed to protect against the first way to break type safety (assuming the BCV is correct, of course). These methods are discussed in more detail below, and for the first three we discuss whether they work on the cards we experimented with.

3.1 CAP File Manipulation

The simplest way to get ill-typed code running on a card is to edit a CAP (Converted APplet) file to introduce a type flaw in the bytecode and install it on the card. Of course, this will only work for cards without on-card BCV and for unsigned CAP files. One can make such edits in the CAP file or – which is easier – in the more readable JCA (Java Card Assembly) files. For example, to treat a `byte` array as a `short` array, it is enough to change a `aload` (byte load) opcode into a `saload` (short load). Such a misinterpreted array type can potentially lead to accessing other applets’ memory as we explain in Sect. 4. To further simplify things, instead of editing JCA or CAP files, one could use some tool for this; ST Microelectronics have developed such a tool, which they kindly let us use.

CAP file editing gives endless possibilities to produce type confusion in the code, including type confusion between references and values of primitive types, which in turn allows C-like pointer arithmetic.

Experiments. As expected, all the cards without on-card BCV installed ill-typed CAP files without problems. Apart from an on-card BCV, simply prohibiting applet loading – common practice on smartcards in the field – or at least controlling applet loading with digital signatures are of course ways to prevent against manipulated CAP files.

3.2 Abusing Shareable Interface Objects

The shareable interface mechanism of Java Card can be used to create type confusion between applets without any direct editing of CAP files, as first suggested in [11].

Shareable interfaces allow communication between applets (or between security contexts, to be precise): references to instances of shareable interfaces can be legally used across the applet firewall. To use this to create type confusion, the trick is to let two applets communicate via a shareable interface, but to compile and generate CAP files for the applets using different definitions of the shareable interface. This is possible because the applets are compiled and loaded separately.

For example, suppose we have a server applet exposing a shareable interface `MyInterface` and a second client applet using this interface. If we produce the CAP file for the server applet using the following interface definition:

```
public interface MyInterface extends Shareable {
    void accessArray(short[] array); } // Server assumes short[]
```

and the CAP file for the client applet using:

```
public interface MyInterface extends Shareable {
    void accessArray(byte[] array); } // Client assumes byte[]
```

then we can make the server applet treat a `byte` array as a `short` array.

One last thing to take care of in this scenario is to circumvent the applet firewall mechanism. Since the server and client applet reside in different contexts, the server does not have the right to access the array it gets from the client. Hence, to make this work the server has to first send its own `byte` array reference to the client and then the client has to send it back to the server through the ill-typed interface. This way the server can run malicious code on its own (in terms of context ownership) data. Now, the shareable interface definition for the server will for instance include:

```
void accessArray(short[] array); // Server assumes short[]
byte[] giveArray();             // Server gives its array to client
```

whereas the one for the client includes:

```
void accessArray(byte[] array); // Client assumes byte[]
byte[] giveArray();             // This array from server is sent back
                                // to the server with accessArray(...)
```

Obviously, this scenario is not limited to confusing `byte` arrays with `short` arrays. Virtually any two types can be confused this way.

We should point out that the client and server applet usually need to be aware of each other and actively cooperate to cause an exploitable type unsoundness. So they both have to be malicious. To the best of our analysis it is not really possible to type-attack an ‘unaware’ server which exports a shareable interface, by crafting a malicious client applet, or vice versa.

Experiments. This method to break the type system worked on all cards without BCV, with the exception of D_211. Card D_211, without on-card BCV, refused to load any code that uses shareable interfaces – for reasons still unclear to us.

Both cards with on-card BCV, C_211A and C_211B, also refuse to install code that uses shareable interfaces, but that is understandable. An on-card BCV may

have a hard time spotting type flaws caused by the use of incompatible interfaces definition, because just performing bytecode verification of an individual applet will not reveal that anything is wrong. So cards C_211A and C_211B resort to a simpler and more extreme approach: they simply refuse to load any code that use shareable interfaces. This clearly avoids the whole issue with type checking such code. (Strictly speaking, one can argue that these cards do not implement the Java Card standard correctly, as there is no mention in the Java Card specification of shareable interfaces being an optional feature.)

3.3 Abusing the Transaction Mechanism

The Java Card transaction mechanism, defined in [9], is probably the trickiest aspect of the Java Card platform. The mechanism has been the subject of investigation in several papers [15], and [4] demonstrated it as a possible source of fault injections on one card. The transaction mechanism allows multiple bytecode instructions to be turned into an atomic operation, offering a roll-back mechanism in case the operation is aborted, which can happen by a card tear or an invocation of the API method `JCSystem.abortTransaction`. The roll-back mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to `null` [9, Sect. 7.6.3].

As pointed out to us by Marc Witteman, it is this last aspect which can be abused to create type confusion: if such references are spread around, by assignments to instance fields and local variables, it becomes difficult for the transaction mechanism to keep track of which references should be nulled out. (This problem is similar to reference tracking for garbage collection, which is normally not supported on Java Cards.) For example, consider the following program:

```
short[] array1, array2; // instance fields
...
    short[] localArray = null; // local variable
    JCSystem.beginTransaction();
        array1 = new short[1];
        array2 = localArray = array1;
    JCSystem.abortTransaction();
```

Just before the transaction is aborted, the three variables `array1`, `array2`, and `localArray` will all refer to the same `short` array created within the transaction. After the call to `abortTransaction`, this array will have been deallocated and all three variables should be reset to `null`.

However, buggy implementations of the transaction mechanism on some cards keep the reference in `localArray` and reset only `array1` and `array2`. On top of this, the new object allocation that happens after the abort method reuses the reference that was just (seemingly) freed. Thus the following code:

```
short[] arrayS; // instance field
byte[] arrayB; // instance field
...
```

```
short[] localArray = null; // local variable
JCSYSTEM.beginTransaction();
    arrayS = new short[1]; localArray = arrayS;
JCSYSTEM.abortTransaction();
arrayB = new byte[10]; // arrayB gets the same reference as arrayS
                    // used to have, this can be tested
if((Object)arrayB == (Object)localArray) ... // this is true!
```

produces two variables of incompatible types, `arrayB` of type `byte[]` and `localArray` of type `short[]`, that hold the same reference, so we have ill-typed code. Again, this trick is not limited to array types.

The root cause of this problem is the subtle ‘feature’ of the transaction mechanism that stack-allocated variables, such as `localArray` in the example above, are *not* subject to roll-back in the event of a programmatically aborted transaction (i.e. a call to `JCSYSTEM.abortTransaction`). Apparently this potential for trouble has been noticed, as the JCRE specification [9, Sect. 7.6.3] explicitly allows a card to mute in the event of a programmatic abort after objects have been created during the transaction.

Experiments. Four cards (B_211, B_221, C_211A, D_211) have a buggy transaction mechanism implementation that allows the type system to be broken in the way described above. Note that an on-card BCV will *not* prevent this attack. Indeed, one of the cards with an on-card BCV, C_211A, is vulnerable to this attack.

The obvious countermeasure against this attack is to correctly implement the clean-up operation for aborted transactions. However, only one of our test cards (B_22) managed to perform a full clean-up correctly.

Another countermeasure against this attack is for cards to mute when a transaction during which objects have been created is programmatically aborted. As mentioned above, this is allowed by the JCRE specifications. Three cards we tested (A_211, A_221, C_211B) implement this option.

3.4 Fault Injections

Finally, fault injections, e.g. by light manipulations, could in theory be used to change the bytecode installed on a card and introduce type flaws.

Fault injections do not provide a very controlled way to change memory, so the chance of getting an exploitable type flaw is likely to be small. However, following the ideas described in [3], for specially tailored code a fault injection can be expected to produce an exploitable type flaw with a relatively high chance.

We did not carry out any experiments with this, since we do not have the hardware to carry out fault injections.

4 Type Attacks on Java Cards

Using the various methods discussed in the previous section, we were able to install ill-typed code on all but one of the smartcards we had (namely card C_211B). We then experimented with various attacks on these cards.

One idea was to exploit type confusion between primitive arrays of different types, a `byte` array with a `short` array, to try and access arrays beyond the array bounds. Another basic idea was to exploit type confusion between an array and an object that was not an array, where there are several opportunities for mischief, such as redefining an array's length – reportedly successful on some cards [11] – or writing object references to perform pointer arithmetic or spoof references. Whether confusion between arrays and other objects can be exploited depends on the exact representation of objects in memory.

4.1 Accessing a Byte Array as a Short Array [Byte as Short Array]

The first attack is to try to convince the applet to treat a `byte` array as a `short` array. In theory this would allow one to read (or write) twice the size of the original `byte` array. For instance, accessing a `byte` array of length 10 as a `short` array size might allow us to access 10 shorts, i.e. 20 bytes, with the last 10 bytes possibly belonging to another applet.

We considered three kinds of byte arrays: global arrays (i.e. the APDU buffer), persistent context-owned arrays, and transient context-owned arrays. There was no difference in behaviour between these different kinds of arrays, i.e. each card gives the same result for all three array types. However, different cards did exhibit different behaviour, as described below.

Cards `C_211A` and `D_211` gave us the result we were hoping for, as attackers. It was possible to read beyond the original array bound. In particular, we managed to access large parts of the CAP file that was later installed on the card. This is clearly dangerous, as it means an applet could read and modify code or data of another applet that was installed later. NB `C_211A` is the card *with* on-card BCV where the buggy transaction mechanism allowed us to run ill-typed code. This highlights the danger of putting all one's trust in an on-card BCV!

Cards from manufacturer B did not let us read a single value from an ill-typed array. Cards `B_211` and `B_221` muted the current card session whenever we tried this, and `B_22` returned a response APDU with status word `6F48`. This suggests that these cards perform runtime type checking (at least enough to detect this particular type confusion). Indeed, all attacks we tried on `B_211` and `B_221` were ineffective in that they always gave this same result, i.e. the cards muted whenever an ill-typed instruction was executed. For card `B_22` some attacks did give more interesting results.

Results on cards from manufacturer A were hardest to understand. Card `A_221` allowed us to run the ill-typed code. However, it does not let us read data outside the original array bounds. What happens is that one byte value is treated as one short value (exactly as if bytes were in fact implemented as shorts in the underlying hardware). For positive byte values each value is prepended with a `00`, for negative values with `FF`:

```
Read as byte[] 00 01 02 03 04 ... 80 81 ...
Read as short[] 0000 0001 0002 0003 0004 ... FF80 FF81 ...
```

Card `A_211` allowed us to run the ill-typed code and reads two subsequent bytes as one short value:

```
Read as byte[] 00 01 02 03 04 05 06 07 ...
Read as short[] 0001 0203 0405 0607 ...
```

However, it was not possible to go outside of the range of the original byte array: even though the presumed short array reports to be of size n , it is only possible to access the first $n/2$ elements, allowing access to the original n bytes. Attempts to access array elements beyond this yielded an `ArrayIndexOutOfBoundsException`.

What appears to be happening on `A_211` is that array bounds checks are done in terms of the physical memory size of the arrays (in bytes), not in terms of the logical size of the arrays (in number of elements). We will call this *physical bounds checking*.

4.2 Accessing an Object as an Array [Object as Array]

Instead of confusing two arrays of different type, one can more generally try to confuse an arbitrary object with an array. This opens the following attack possibilities.

4.2.1 Fabricating Arrays

Witteman [11] describes a type attack which exploits type confusion between an array and an object of the following class `Fake`:

```
public class Fake { short len = (short)0x7FFF;}
```

The attack relies on a particular representation of arrays and objects in memory: for the attack to succeed, the length field of an array has to be stored at the same offset in physical memory as the `len` field of a `Fake` object. If we can then trick the VM in treating a `Fake` object as an array, then the length of that array would be `0x7FFF`, giving access to 32K of memory. The fact that we can access the `len` field through the object reference could allow us to set the array length to an arbitrary value.

Although setting the length of the array was not possible as such on the cards we tested, this attack gave us interesting results nevertheless.

As before, cards `B_211` and `B_221` refused to execute the ill-typed code, as in all other attack scenarios. Card `A_211` also refused to execute the code, returning the `6F00` status word. Apparently `A_211` has some runtime checks that prevent type confusion between objects and arrays.

On the cards where running our exploit code was possible (`A_221`, `B_22`, `C_211A`, `D_211`), the object representation in memory prevents us from manipulating the array length. Still, we noticed two different behaviours. For cards `A_221` and `B_22`, the length of the “confused” array indicates the number of fields in the object, i.e. an instance of the `Fake` class gives an array of length 1 containing the element `0x7FFF`. For the two other cards, `C_211A` and `D_211`, the length of the confused array has some apparently arbitrary value: the length

is not the number of instance fields, but it probably represents some internal object data. For `C_211A` this value is large enough (actually negative when the length is interpreted as a signed `short`) to freely access any forward memory location on the card.

A slight modification of this attack allows us to read and write object references directly as we describe next.

4.2.2 Direct Object Data Access

The results of the previous attack suggests that it would be possible to treat object fields as array elements on cards `A_221`, `B_22`, `C_211A`, and `D_211`. Reference fields could then be read or written as numerical values, opening the door to pointer arithmetic.

This is indeed possible for all these cards. For example, an instance of this class:

```
public class Test {
    Object r1 = new Object();
    short s1 = 10; }
```

when treated as a short array `a` on card `A_221` gives the following array contents:

```
a.length:    2          # of fields, read only
a[0]:        0x09E0     field r1, read/write
a[1]:        0x000A     field s1, read/write
```

By reading and writing the array element `a[0]` it was possible to directly read and write references. Similar results were obtained on the three other cards (`B_22`, `C_211A`, `D_211`), although in the case of `C_211A` we did not manage to effectively write a reference.

Pursuing this attack further we tried two more things: switching references around and spoofing references.

4.2.3 Switching References [Switch]

Once we have a direct access to a reference we can try to replace it (by direct assignments) with another reference, even if these have incompatible types. Our test results show that if the new value is a valid reference (i.e. existing reference) this is perfectly possible. Assume, for example, that we have these two field declarations in some class `C`:

```
MyClass1 c1 = new MyClass1();
MyClass2 c2 = new MyClass2();
```

Accessing an object of class `C` as an array, we should be able to swap the values of `c1` and `c2`. This in turn introduces further type confusion: field `c1` points to a reference of type `MyClass2` and field `c2` to a reference of type `MyClass1`.

Two cards, `A_221` and `B_22`, allowed us to do it. It was possible to read instance fields of such switched references, but only as long as the accessed fields stayed within the original object bounds. This suggests that these cards perform dynamic bounds checks when accessing instance fields, analogous to the dynamic bounds checks when accessing array elements. We will call this *object bounds*

checking. Indeed, given the similarity of memory layout for arrays and other objects on these card, the code for accessing instance fields and array elements might be identical. Such object bounds checking prevents reference switching from giving access beyond original object bounds, and hence prevents access to another applet's data.

Another way in which reference switching might give access to another applet would be setting a field in one applet to point to an object belonging to another applet. However, here the firewall checks prevent illegal access via such a reference.

Also, methods can be called on the switched references, as long as calling such methods did not cause object bounds violations or referring to stack elements below the stack boundary.

4.2.4 AID Exploitation [AID]

The possibility to switch references could be abused to manipulate system-owned AID objects. An AID (Applet Identifier) object is effectively a byte sequence that is used to identify applets on a card. An AID object has a field which points to a byte array that stores the actual AID bytes. Changing the value of this field would change the value of an AID, whereas AIDs are supposed to be immutable.

An obstacle to this attack might be the firewall mechanism; indeed, if we try to change the field of a system-owned AID object from an applet this is prevented by the firewall. However, if we access the AID object as an array, then on cards `A_221` and `B_22` we could change the values of system-owned AIDs. This has serious consequences: a hostile applet can corrupt the global AID registry, and try to impersonate other applets. This attack is a much stronger version of the one described in e.g. [7].

Because of the privileged nature of system-owned AID objects – they are JCRE entry points – further exploits might be possible.

4.2.5 Spoofing References [Spoof]

Going one step further than switching references, we tried spoofing references, i.e. creating a reference from a numerical value by assigning a `short` value to a reference.

Any way we tried this, cards `A_221`, `B_22`, and `C_211A` refused to use such references: the card session was muted or an exception was thrown.

Card `D_211`, an older generation card, did let us spoof references. It was possible to write a small applet that effectively let us “read” any reference from the card by using the techniques we just described and a little bit of CAP file manipulation trickery. By “read” we mean that it is possible to get *some* value that supposedly resides at the memory address indicated by the reference. However, composing a sequence of such reads (going through all the possible reference values) did not really give a valid memory image of the card. That is, it was not possible to recognise parts of bytecode that should be on the card, or any applet data we would expect to find.

Cards can detect spoofing of references by keeping track of additional data in their representation of objects or references in memory and refusing to operate on (references to) objects if this data is not present or corrupted.

For instance, analysing our failed attempts to spoof references on `A_221` we noticed that each allocated object (even the simplest one) takes up at least 8 bytes of memory. That is, the values of references to subsequently allocated objects, when read as numerical values, differ by at least eight. An array of length 1 would even take up 16 bytes. It is clear that two bytes are used to store the number of fields (or array length, in case of an array). However, it is not clear what the other six bytes (or more in case of arrays) are used for: it will include information about the runtime type and the firewall context, but it could also contain additional checksums to check the integrity of a reference. If it would be possible to reconstruct the structure of this data (difficult because of the number of combinations to try) we believe constructing a fake reference could be considered a possibility, although an unlikely one.

We also tested references of arrays of different memory type (persistent and transient). The values of references to different kinds of arrays seem to be ‘next to each other’, which would indicate that the value of the reference has little to do with the actual memory address. Apparently there is an additional mechanism in the VM to map these reference to physical addresses.

4.3 More Type Confusion Attacks

Obviously, the attacks we have just described do not exhaust all possibilities. Many more are possible. For example, by changing the number of parameters in the `shareable` method one could try to read data off the execution stack, but this did not succeed on any of our cards.

Another example is that it is possible to reverse the type confusion between arrays and objects, and access an array as an object, with the aim to try accessing an array beyond its array bounds. Such an ‘array as object’ attack produced similar results as the object as array attack in Sect. 4.2.2, except that it was also possible on `A_211`, albeit harmless in the sense that it did not allow access beyond the original array bounds there.

5 Dynamic Countermeasures

Our experiments show that some VMs are much more defensive than others when it comes to executing ill-typed code. The Java Card specifications leave a lot of freedom for defensive features in the implementation of a VM. As long as the executed code is well-typed the defensive features should go undetected; the specifications are meant to guarantee that well-typed code executing on different smartcards always results in the same behaviour. However, ill-typed code is effectively out of scope as far as the Java Card specifications are concerned; when running ill-typed code as we have done, there are *no* guarantees that the same behaviour is observed on different cards, and additional defences can come to light. Below we give an overview of possible dynamic runtime checks a VM might implement.²

² The fact that Java Cards take so many clock cycles for each individual bytecode instruction [10] already suggests that Java Cards do quite a lot of runtime checks.

Runtime Type Checking. Two cards from manufacturer B, cards B_211 and B_221, appear to do runtime type checking, making them immune to all ill-typed code we tried. Card A_211 also performs enough runtime type checks to make it immune to all our attacks. Still, because we were able to confuse byte and short arrays, albeit without being able to do actual harm, card A_211 apparently does not do complete type checking at runtime.

Object Bounds Checking. Any VM is required to do runtime checks on array bounds when accessing array elements. A VM could do some similar runtime checks of object bounds when accessing instance fields and invoking methods on objects that are not arrays, if it records the ‘size’ – the number of fields – of each object in the same way it records the length of an array. In the conversion of bytecode into CAP file format, names are replaced by numbers – 0 denotes the first field, 1 the second, etc. – which makes such a check easy to implement.

Two of the cards appear to do object bounds checking, namely A_221 and B_22, as explained in Sect. 4.2.3.

Physical Bounds Checks. Bounds checks on arrays (or objects) can be done using the ‘logical’ size of an array (i.e. its length and the array index), but can also be done using the ‘physical’ size of the array contents in bytes and the actual offset of an entry. For example, the contents of a `short` array `a` of length 10 takes up 20 bytes. When accessing `a[i]`, one could do a runtime check to see if $0 \leq i \leq 10$, or a runtime check to see if $0 \leq 2*i \leq 20$. If the VM uses the physical offset $2*i$ to look up the entry, one may opt for the latter check.

Our experiments suggests that card A_211 performs physical bounds checks, as explained in Sect. 4.1.

Firewall Checks. Firewall checks have to be done at runtime.³ Our successful attacks on C_211A and D_211 by confusing `byte` and `short` arrays in Sect. 4.1 as well as two successful AID object exploits (A_221, B_22) demonstrate that the firewall does not really provide defence-in-depth in the sense that it can compensate for the absence of a bytecode verifier.

Still, in *some* attacks the firewall is clearly a formidable second line of defence. For instance, attacks where we try to switch references are essentially harmless if the firewall prevents us from accessing object belonging to other contexts: at best an applet could then attack its own data, which is not an interesting attack. However, as the AID attack show, the firewall does not prevent all such attacks, as by accessing object as arrays we might be able to circumvent firewall checks. This shows that firewall implementations are not defensive, in the sense they do not make additional checks to catch type confusion, but then the specification of the firewall does not require them to be defensive.

It is conceivable that a defensive implementation of the firewall could prevent the attacks on C_211A and D_211 described in Sect. 4.1, namely if firewall checks

³ A system to statically check most of the firewall access rules is proposed in [2]. However, performing checks statically, at load time, is not necessarily an improvement over doing them at runtime, as our results with bytecode verification show.

are performed on the ‘physical’ rather than the ‘logical’ level, as discussed above for array bound checks. Checking firewall rules at the ‘physical level’ would require that the VM can somehow tell the context for every memory cell, not just every reference. One way to do this would be to allocate a segment of memory for each security context, and then use old-fashioned segmentation as in operating systems as part of enforcement of the firewall. We found no evidence that any cards do this.

Integrity Checks in Memory. Our experiments with spoofing references suggest that most cards provide effective integrity checks on references. To perform dynamic checks for array bounds, downcasting, and the firewall, the VM has to record some meta-data in the memory representation of objects, such as array lengths, runtime types and the context owning an object. Clearly, by recording and checking additional meta-data in objects (and possibly references) the VM could detect and prevent switching or spoofing of references.

6 Discussion

Table 1 summarises the result of Sect. 3, and shows which of the ways to get ill-typed code succeeded on which of the cards.

CAP file manipulation (CAP) and Shareable Interface Objects (SIO) did not succeed on C_211A and C_211B, because the on-card BCV does not allow ill-typed code or any code that uses shareable interfaces. The loader on D_211 also refuses to load code that uses shareable interfaces.

Abusing the transaction mechanism works on cards B_211, B_221, C_211A, and D_211, which indicates that the implementation of the transaction mechanism is faulty when it comes to clearing up after aborted transactions.

Card C_211B was the only one on which we were unable to load any ill-typed code.

However, being able to load ill-typed code on a card does not guarantee success (from an attacker’s point of view), as defensive features of the VM can still prevent the loaded code from doing any harm or executing at all. Things do *not* necessarily degenerate to the hopeless situation one has in C/C++, where you can basically do anything with malicious code.

Table 2 summarises the results of Sect. 4, listing which attacks succeeded in running and doing harm.

Table 1. Loading ill-typed code

	A_211	A_221	B_211	B_22	B_221	C_211A	C_211B	D_211
CAP file manipulation [3.1]	+	+	+	+	+	–	–	+
Abusing shareable interfaces [3.2]	+	+	+	+	+	–	–	–
Abusing transactions [3.3]	–	–	+	–	+	+	–	+
Static protection						BCV	BCV	CL

BCV – On-card static bytecode verifier

CL – class loader disallowing use of shareable interfaces

Table 2. Executing ill-typed code. No information is included for card C_211B, since we could not load ill-typed code on it.

	A_211	A_221	B_211	B_22	B_221	C_211A	D_211
Dynamic protection	PBC	OBC	RTC	OBC	RTC		
byte as short array [§4.1]	✓	✓	–	–	–	↯	↯
Object as array [§4.2.1]	–	✓	–	✓	–	↯	↯
Reference switching [§4.2.3]	–	✓	–	✓	–	–	NA
Reference switching in AIDs [§4.2.4]	–	↯	–	↯	–	–	NA
Reference spoofing [§4.2.5]	–	–	–	–	–	–	↯
Array as object [§4.3]	✓	✓	–	✓	–	↯	↯

– impossible, ✓ possible but harmless, ↯ possible and harmful

PBC – Physical Bounds Checks, OBC – Object Bounds Checks,

RTC – Runtime Type Checking, NA – Not attempted

Some attacks can do real damage. For cards C_211A and D_211 two different attacks are possible to access another applet’s data, namely accessing a `byte` as a `short` array and accessing an array as an object. The latter attacks allows unrestricted forward memory access on C_211A. Switching references on A_221 and B_22 is possible but mostly harmless, since the firewall prevents access to data of another applet. The notable exception is using this attack to access the internals of AID objects, where the attack becomes harmful as it allows a hostile applet to alter any system-owned AIDs and redefine the entire AID registry of the card. Spoofing references on D_211 also allowed unrestricted memory access; even though the memory still seemed to be scrambled, and we could not exploit access to it in a meaningful way, we do regard this as dangerous.

One interesting conclusion is that having an on-card BCV is not all that it is cracked up to be: one of the vulnerable cards we identified has an on-card BCV. A single bug, in this case in the transaction mechanism, can completely undermine the security provided by the on-card BCV⁴. Also, cards with an on-card BCV rule out any use of Shareable Interfaces, which in retrospect is understandable, but we never realised this before we tried.

As a defensive mechanism, runtime type checking is therefore probably a more robust protection mechanisms than a static BCV. Indeed, another interesting conclusion of our work is that runtime defensive mechanisms go a long way to protect against ill-typed code, as results with card A_211, B_211, and B_221 show. The obvious disadvantage of runtime checks is the decreased performance of the JVM. However, we did not notice any considerable performance differences between our cards. More factors play a role in smartcard performance (such as the underlying hardware), so more testing would be required to establish what is the actual impact of runtime checks on performance.

We should repeat that the vulnerabilities found are of course only a problem on cards allowing the installation of additional applets. On most, if not all, Java

⁴ One hopes that Sun’s Technology Compatibility Kit (TCK) for Java Card includes test cases to detect this bug. Unfortunately, the TCK is not public so we cannot check that it does.

Card smartcards in the field, post-issuance download of additional applets will be disabled or at least restricted by digital signatures. Still, for security evaluations it can be extremely useful (and cost-saving) to have confidence in the fact that there are no ways for different applets to affect each other's behaviour, except when explicitly allowed by shareable interfaces.

Acknowledgements. Thanks for Marc Witteman of Riscure and Olivier van Nieuwenhuyze and Joan Daemen at STMicroelectronics for their insights. We also thank Riscure for access to their extensive Java Card test suite, and STMicroelectronics for access to their CAP file manipulation tool.

The work of Wojciech Mostowski is supported by Sentinels, the Dutch research programme in computer security, which is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

References

1. Beckert, B., Mostowski, W.: A program logic for handling Java Card's transaction mechanism. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 246–260. Springer, Heidelberg (2003)
2. Dietl, W., Müller, P., Poetzsch-Heffter, A.: A Type System for Checking Applet Isolation in Java Card. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 129–150. Springer, Heidelberg (2005)
3. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: IEEE Symposium on Security and Privacy, pp. 154–165 (2003)
4. Hubbers, E., Mostowski, W., Poll, E.: Tearing Java Cards. In: Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22 (2006)
5. Marché, C., Rousset, N.: Verification of Java Card applets behavior with respect to transactions and card tears. In: Proc. Software Engineering and Formal Methods (SEFM), Pune, India. IEEE Computer Society Press, Los Alamitos (2006)
6. McGraw, G., Felten, E.W.: Securing Java. Wiley, Chichester (1999), <http://www.securingjava.com/>
7. Montgomery, M., Krishna, K.: Secure object sharing in Java Card. In: Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard 1999), Chicago, Illinois, USA, May 10–11 (1999)
8. Mostowski, W., Poll, E.: Testing the Java Card Applet Firewall. Technical Report ICIS-R07029, Radboud University Nijmegen (December 2007), <https://pms.cs.ru.nl/iris-diglib/src/icis-tech-reports.php>
9. Sun Microsystems, Inc. Java Card 2.2.2 Runtime Environment Specification (March 2006), <http://www.sun.com>
10. Vermoen, D.: Reverse engineering of Java Card applets using power analysis. Technical report, TU Delft1 (2006), http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf
11. Witteman, M.: Java Card security. Information Security Bulletin 8, 291–298 (2003)

Static Program Analysis for Java Card Applets

Vasilios Almaliotis¹, Alexandros Loizidis¹, Panagiotis Katsaros¹,
Panagiotis Louridas², and Diomidis Spinellis²

¹ Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece

{valmalio, aloizidi, katsaros}@csd.auth.gr

² Department of Management Science and Technology,
Athens University of Economics and Business,

Patision 76, 104 34 Athens, Greece

{louridas, dds}@aueb.gr

Abstract. The Java Card API provides a framework of classes and interfaces that hides the details of the underlying smart card interface, thus relieving developers from going through the swamps of microcontroller programming. This allows application developers to concentrate most of their effort on the details of application, assuming proper use of the Java Card API calls regarding (i) the *correctness of the methods' invocation targets and their arguments* and (ii) *temporal safety*, i.e. the requirement that certain method calls have to be used in certain orders. Several characteristics of the Java Card applets and their multiple-entry-point program structure make it possible for a potentially unhandled exception to reach the invoked entry point. This contingency opens a possibility to leave the applet in an unpredictable state that is potentially dangerous for the application's security. Our work introduces automatic static program analysis as a means for the early detection of misused and therefore dangerous API calls. The shown analyses have been implemented within the FindBugs bug detector, an open source framework that applies static analysis functions on the applet bytecode.

Keywords: Java Card, static program analysis, temporal safety.

1 Introduction

Static analysis has the potential to become a credible means for automatic verification of smart card applications that by definition are security critical. This work explores the adequacy of the FindBugs open source framework for the static verification of correctness properties concerned with the API calls used in Java Card applications.

The Java Card API provides a framework of classes and interfaces that hides the details of the underlying smart card interface, thus allowing developers to create applications, called applets, at a higher level of abstraction. The Java Card applet life cycle defines the different phases that an applet can be in. These phases are (i) loading, (ii) installation, (iii) personalization, (iv) selectable, (v) blocked and (vi) dead. A characteristic of Java Card applets is that many actions can be performed only when an applet is in a certain phase. Also, contrary to ordinary Java programs that have a single `main()` entry point, Java Card applets have several entry points, which are

called when the card receives various application (APDU) commands. These entry points roughly match the mentioned lifetime phases.

In a Java Card, any exception can reach the top level, i.e. the applet entry point invoked by the Java Card Runtime Environment (JCRE). In this case, the currently executing command is aborted and the command, which in general is not completed yet, is terminated by an appropriate status word: if the exception is an *ISOException*, the status word is assigned the value of the reason code for the raised exception, whereas in all other cases the reason code is 0x6f00 corresponding to “no precise diagnosis”.

An exception in an applet’s entry point can reveal information about the behavior of the application and in principle it should be forbidden. In practice, whereas an *ISOException* is usually explicitly thrown by the applet code using `throw`, a potentially unhandled exception is *implicitly raised when executing an API method call that causes an unexpected error*. This may result in leaving the applet in an unpredicted and ill state that can possibly violate the application’s security properties.

The present article introduces a static program analysis approach for the detection of misused Java Card method calls. We propose the use of appropriate bug detectors designed for the *FindBugs* static analysis framework. These bug detectors will be specific to the used API calls and will check (i) the correctness of the methods’ invocation target and their arguments and (ii) temporal safety in their use.

We introduce the two bug detectors that we developed by applying interprocedural control flow based and dataflow analyses on the Java Card bytecode. Then, we discuss some recent advances in related theory that open new prospects to implement sufficiently precise property analyses.

Our proposal addresses the problem of unhandled exceptions based on *bug detectors that in the future may be supplied by Java Card technology providers*. Applet developers will check their products for correct use of the invoked API calls, without having to rely on advanced formal techniques that require highly specialized analysis skills and that are not fully automated.

Section 2 provides a more thorough view of the aims of this work and reviews the related work and the basic differences of the presented approach. Section 3 introduces static analysis with the *FindBugs* framework. Section 4 presents the work done on the static verification of Java Card API calls and section 5 reports the latest developments that open new prospects for implementation of efficient static analyses that do not compromise precision. The paper ends with a discussion on our work’s potential impact and comments interesting future research prospects.

2 Related Work on Static Verification of Java Card Applets

Our work belongs to a family of static verification techniques that *in general do not guarantee sound and complete analysis*. This means that there is no guarantee that we will find all property violations and also we cannot guarantee the absence of *false positives*. However, our bug detectors may implement advanced static analyses that eliminate false negatives and minimize false positives (Section 5).

In related work, this sort of analysis cannot be compared with established formal approaches that have been used successfully in static verification of Java Card applets:

JACK [1], KeY [2], Krakatoa [3], Jive ([4], [5]) and LOOP ([6], [7]). These techniques aim in precise program verification and they are *not fully automated*. Moreover, they require highly specialized formal analysis skills for the applet developer.

The most closely related work is the proposal published in [8], where the authors perform Java Card program verification using the ESC/Java (2) tool. This tool proves *correctness of specifications* at compile time, without requiring the analyst to interact with the back-end theorem prover (Simplify). Similarly to our approach, the provided analysis is neither sound nor complete, but has been found effective in proving absence of runtime exceptions and in verifying relatively simple correctness properties.

In contrast with [8], our proposal for static program analysis is not based on source code annotations. This reduces the verification cost to the applet developers, since they do not have to make explicit all implicit assumptions needed for correctness (e.g. the non-nullness of `buf` in many Java Card API calls). Instead of this, the static analyses of FindBugs are implemented in the form of tool plugins that may be distributed together with the used Java Card Development kit or by an independent third party. Applet developers use the bug detectors as they are, but they can also extend their open source code in order to develop bug detectors for custom properties. Note that in ESC/Java (2), user-specified properties assume familiarization, (i) with the Java Modeling Language (JML), (ii) with the particular “design by contract” specification technique and (iii) with the corresponding JML based Java Card API specification [9]. On the other hand, the development of new FindBugs bug detectors assumes only Java programming skills that most software engineers already have.

User defined bug detectors may be based on an initial set of basic bug detectors concerned with (i) the correctness of the API calls’ invocation target and their arguments and (ii) the temporal safety in their use. This article is inspired by the ideas presented in [10]. However, the focus on the Java Card API is not the only one difference between these two works. The static analysis of [10] is based on stateless calls to a library that reflects the API of interest. Violations of temporal safety for the analyzed API calls, however, can be detected only by statefull property analysis that spans the whole applet or even multiple applets in the same or different contexts. As we will see in next sections, FindBugs static analyses are applied by default to individual class contexts and this is one of the restrictions we had to overcome.

3 Static Analysis with the FindBugs Framework

FindBugs [11] is a tool and framework that applies static analyses on the Java (Java Card) bytecode in order to detect *bug patterns*, i.e. to detect “places where code does not follow correct practice in the use of a language feature or library API” [12]. In general, FindBugs bug detectors behave according to the Visitor design pattern: each detector visits each class and each method in the application under analysis. The framework comes with many analyses built-in and classes and interfaces that can be extended to build new analyses. In our work, we exploit the already provided *intra-procedural control flow analysis* that transforms the analyzed bytecode into *control flow graphs* (CFGs) that support the property analyses and dataflow analyses presented in next sections.

The bug pattern detectors are implemented using the Byte Code Engineering Library (BCEL) [13], which provides infrastructure for analyzing and manipulating Java class files. In essence, BCEL offers to the framework data types for inspection of binary Java (Java Card) classes. One can obtain methods, fields, etc. from the main data types, `JavaClass` and `Method`. The project source directories are used to map the reported warnings back to the Java source code.

Bug pattern detectors are packaged into *FindBugs plugins* that can use any of the built-in FindBugs analyses and in effect extend the provided FindBugs functionality without any changes to its code. A plugin is a jar file containing detector classes and analysis classes and the following meta-information: (i) the plugin descriptor (`findbugs.xml`) declaring the bug patterns, the detector classes, the *detector ordering constraints* and the analysis engine registrar, (ii) the human-readable messages (in `messages.xml`), which are the localized messages generated by the detector. Plugins are easily activated in the developer's FindBugs installation by copying the jar file into the proper location of the user's file system.

FindBugs applies the loaded detectors in a series of `AnalysisPasses`. Each pass executes a set of detectors selected according to declared detector ordering constraints. In this way, FindBugs distributes the detectors into `AnalysisPasses` and forms a complete `ExecutionPlan`, i.e., a list of `AnalysisPasses` specifying how to apply the loaded detectors to the analyzed application classes. When a project is analyzed, FindBugs runs through the following steps:

1. Reads the project
2. Finds all application classes in the project
3. Loads the available plugins containing the detectors
4. Creates an execution plan
5. Runs the FindBugs algorithm to apply detectors to all application classes

The basic FindBugs algorithm in pseudo-code is:

```

for each analysis pass in the execution plan do
  for each application class do
    for each detector in the analysis pass do
      apply the detector to the class
    end for
  end for
end for

```

All detectors use a global cache of *analysis objects* and *databases*. An analysis object (accessed by using a `ClassDescriptor` or a `MethodDescriptor`) stores facts about a class or method, for example the results of a null-pointer dataflow analysis on a method. On the other hand, a database stores facts about the entire program, e.g. which methods unconditionally dereference parameters. All detectors implement the `Detector` interface, which includes the `visitClassContext` method that is invoked on each application class. Detector classes (i) request one or more analysis objects from the global cache for the analyzed class and its methods, (ii) inspect the gathered analysis objects and (iii) report warnings for suspicious situations in code. When a `Detector` is instantiated its constructor gets a reference to a `BugReporter`. The `Detector` object uses the associated `BugReporter`, in

order to emit warnings for the potential bugs and to save the detected bug instances in `BugCollection` objects for further processing.

4 Static Verification of Java Card API Calls

The test cases for the bug detectors shown here were derived from an electronic purse applet developed for the purposes of this work. The electronic purse applet adds or removes units of digital currency and stores the personal data of the card owner. Moreover, there is also a bonus applet that interacts with the electronic purse for crediting the bonus units corresponding to the performed transactions. The two applets lie in separate contexts and communicate data to each other through a shareable interface. Both applets are protected by their own PINs. They are accessed through the Java Card Runtime Environment (JCRC) that invokes the `process` method, which in turn invokes the methods corresponding to the inputted APDU commands.

PurseApplet	BonusApplet
+ credit	+ changeUserPIN
+ debit	+ eraseBonus
+ foreignDebit	+ getBonus
+ getAccountNumber	+ makePurchase
+ getBalance	+ subtractBonus
+ getUserAddress	+ validateUserPIN
+ getUserName	
+ getUserSurname	
+ setAccountNumber	
+ setUserAddress	
+ setUserName	
+ setUserSurname	
+ setUserPIN	
+ validateUserPIN	

Fig. 1. Public members of the PurseApplet and the BonusApplet

4.1 Bug Detectors for the Temporal Safety of Java Card API Calls

Bug detectors for the temporal safety of API calls use a control flow graph (CFG) representation of Java methods to perform static verification that either exploits the builtin dataflow analyses or is based on more sophisticated user-defined analyses. The following pseudo-code reflects the functionality of the `visitClassContext()` method of a typical CFG-based detector.

```

for each method in the class do
  request a CFG for the method from the ClassContext
  request one or more analysis objects on the method from the ClassContext
  for each location in the method do
    get the dataflow facts at the location
    inspect the dataflow facts

```

```

if a dataflow fact indicates an error then
    report a warning
end if
end for
end for

```

The basic idea is to visit each method of the analyzed class in turn, requesting some number of analysis objects. After getting the required analyses, the detector iterates through each *location* in the CFG. A location is the point in execution just before a particular instruction is executed (or after the instruction, for backwards analyses). At each location, the detector checks the dataflow facts to see if anything suspicious is going on. If suspicious facts are detected at a location the detector issues a warning.

Temporal safety of API calls concerns rules about their ordering that are possibly associated with constraints on the data values visible at the API boundary. Temporal safety properties for the Java Card API are captured in appropriate state machines that recognize finite execution traces with improper use of the API calls. Figure 2 introduces the state machine for a Java Card applet bug raising an APDUException for improper use of the `setOutgoing()` call.

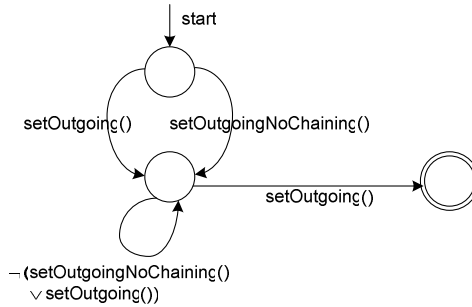


Fig. 2. Illegal use of short `setOutgoing()` corresponding to a Java Card APDUException

Bug detectors for temporal safety of API calls track *the state of the property* and at the same time track the so-called *execution state*, i.e. the values of all program variables. Accurate tracking of the execution state can be very expensive, because this implies tracking every branch in the control-flow, in which the values of the examined variables differ along the branch paths. The resulted search space may grow exponentially or even become infinite.

For the property of Figure 2 we developed the *path-insensitive bug detector*, shown in this section, to explore the suitability of the FindBugs framework for the static verification of Java Card applets. The more precise *path-sensitive analyses* rely on the fact that for a particular property to be checked, it is likely that most branches in the code are not relevant to the property, even though they affect the execution state of the program. Detectors of this type may be based on heuristics that identify the relevant branches and in this way they reduce the number of potential false positives. Recent advances in path-sensitive static analyses and their applicability in the FindBugs framework are discussed in section 5.

In any applet, it is possible to access an APDU provided by the JCRE, but it is not possible to create new APDUs. This implies that all calls to `setOutgoing()` in a single applet are applied to the same APDU instance and this fact eliminates the need to check the implicit argument of the `setOutgoing()` calls. The developed detectors take into account two distinct cases of property violation:

1. *Intraprocedural property violations* are detected by simple *bytecode scanning* that follows the states of the property state machine (Figure 2)
2. *Interprocedural property violations* are detected by extending the CFG based and *call graph analysis* functions provided in the Findbugs framework.

More precisely, the `InterCallGraph` class we developed makes it possible to construct call graphs including calls that span different class contexts. This extension allowed the detection of nested method calls that trigger the state transitions of Figure 2 either by direct calls to `setOutgoing()` or by nested calls to methods causing reachability of the final state. The following is the pseudo-code of the path-insensitive interprocedural analysis.

```

request the call graph of the application classes
for each method in the call graph do //mark methods with setOutgoing() call
    if method contains setOutgoing() then
        add method to the black list
    end if
end for
for each method in the class do //mark methods with nested black method call(s)
    start a Depth First Search from the corresponding graph node:
    if method of the node is in the black list then
        add method to the gray list
        if final state of Fig. 2 is reached then
            report the detected bug
        end if
    end if
end for
for each method in the class do //detect property violation caused in a loop
    request a CFG for the method
    check if method has loop, enclosing call of setOutgoing() or a gray method
end for

```

Finally, the methods' CFGs are inspected for loops enclosing method calls that do not cause reachability of the final state by themselves, but they result in a property violation when encountered in a loop. Figure 3 shows the bytecode patterns matching the use of a loop control flow in a CFG. Unhandled exception violations are detected by looking for an *exception thrower block* preceding the instruction by which we reach the final state (Figure 4). Access to an *exception handler block* (if any) is possible through a *handled exception edge*. In FindBugs, method `isExceptionHandlerThrower()` detects an exception thrower block and method `isExceptionHandlerEdge()` determines whether a CFG edge is a handled exception edge.

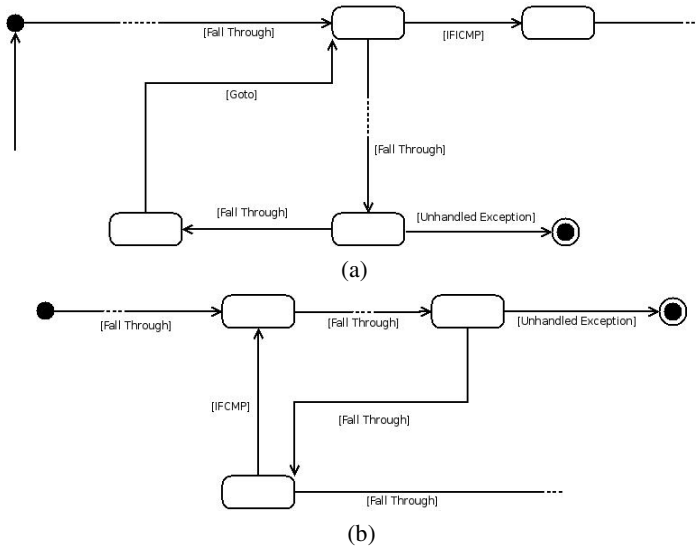


Fig. 3. CFG patterns with basic blocks corresponding to (a) for/while and (b) do . . . while loop

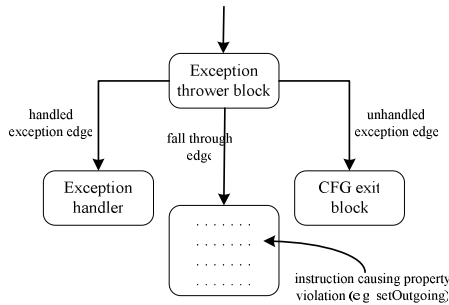


Fig. 4. CFG pattern to find unhandled exception edges

Figure 5 demonstrates how the detector responds in two different property violation cases. In the first case, the client applet named *PurseClientApplet* calls *setOutgoing()* and subsequently invokes the method *getUserAddress()* of the *PurseApplet* thus causing the detected property violation. The second case concerns a property violation caused by a call to *setUserAddress()* in a for loop.

4.2 Bug Detectors for the Correctness of the Called Methods' Arguments

Dataflow analysis is the basic means to statically verify the correctness of the called methods' arguments. Its basic function is to estimate conservative approximations about *facts* that are true in each location of a CFG. Facts are mutable, but they have to form a lattice. The *DataflowAnalysis* interface shown in Figure 6 is the supertype for all concrete dataflow analysis classes. It defines methods for creating,

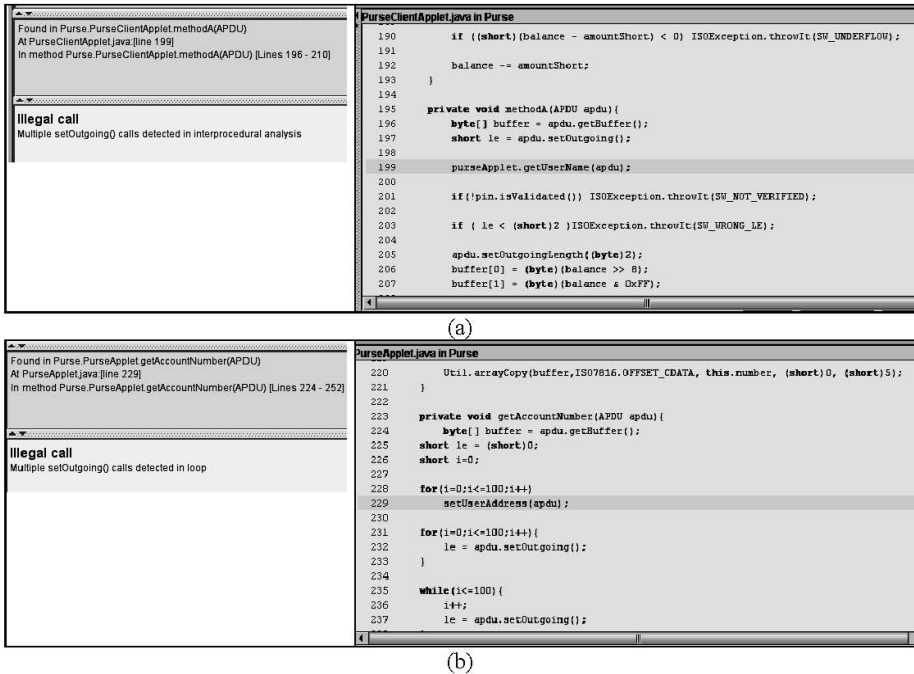


Fig. 5. Illegal use of `setOutgoing()` detected (a) in interprocedural analysis and (b) within a loop via call to another method

copying, merging and transferring dataflow facts. Transfer functions take dataflow facts and model the effects of either a basic block or a single instruction depending on the implemented dataflow analysis. Merge functions combine dataflow facts when control paths merge. The `Dataflow` class and its subclasses implement: (i) a dataflow analysis algorithm based on a CFG and an instance of `DataflowAnalysis`, (ii) methods providing access to the analysis results.

We are particularly interested for the `FrameDataflowAnalysis` class that forms the base for analyses that model values in local variables and operand stack. Dataflow facts for derived analyses are subclasses of the class `Frame`, whose instances represent the Java stack frame at a single CFG location. In a Java stack frame, both stack operands and local variables are considered to be “slots” that contain a single symbolic value.

The built-in frame dataflow analyses used in static verification of the called methods’ arguments are:

- The `TypeAnalysis` that performs type inference for all local variables and stack operands.
- The `ConstantAnalysis` that computes constant values in CFG locations.

- The `IsNullValueAnalysis` that determines which frame slots contain definitely-null values, definitely non-null values and various kinds of conditionally-null or uncertain values.
- The `ValueNumberAnalysis` that tracks the production and flow of values in the Java stack frame.

The class hierarchy of Figure 6 and the mentioned built-in dataflow analyses form a generic dataflow analysis framework, since it is possible to create new kinds of dataflow analyses that will use as dataflow facts objects of user-defined classes.

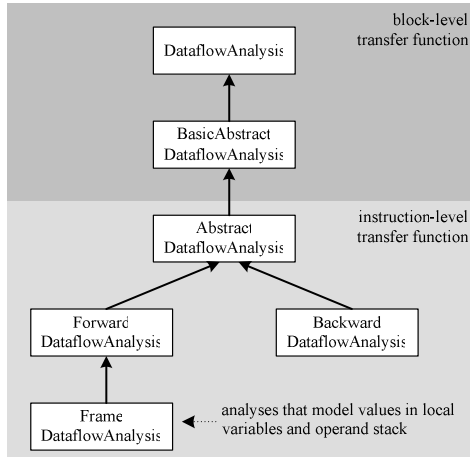


Fig. 6. FindBugs base classes for dataflow analyses

A bug detector exploits the results of a particular dataflow analysis on a method by getting a reference to the `Dataflow` object that was used to execute the analysis. There is no direct support for interprocedural analysis, but there are ways to overcome this shortcoming. More precisely, analysis may be performed in multiple passes. A first pass detector will compute *method summaries* (e.g. method parameters that are unconditionally dereferenced, return values that are always non-null and so on), without reporting any warnings and a second pass detector will use the computed method summaries as needed. However, this approach excludes the implementation of *context sensitive interprocedural analyses* like the ones explored in Section 5.

In the following paragraphs, we present a bug detector for unhandled exceptions concerned with the correctness of arguments in method calls. Consider the following method:

```
short arrayCopy( byte[] src,  short srcOff,
                 byte[] dest, short destOff, short length)
```

A `NullPointerException` is raised when either `src` or `dest` is null. Also, when the copy operation accesses data outside the array bounds the `ArrayIndexOutOfBoundsException` is raised. This happens either when one of the parameters `srcOff`,

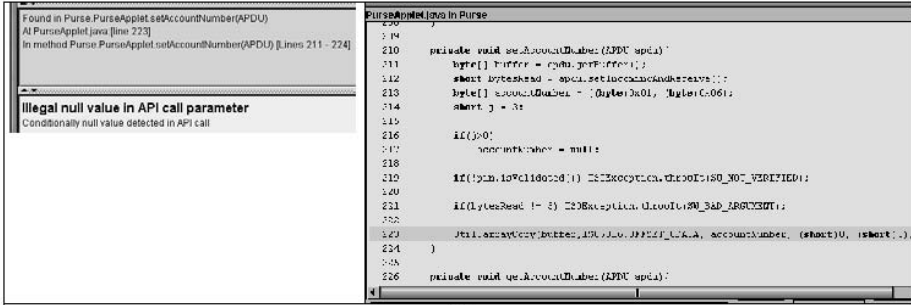
`destOff` and `length` has a negative value or when `srcOff+length` is greater than `src.length` or when `destOff+length` is greater than `dest.length`. We provide the pseudo-code of the `visitClassContext()` method for the detector of unhandled exceptions raised by invalid `arrayCopy` arguments:

```

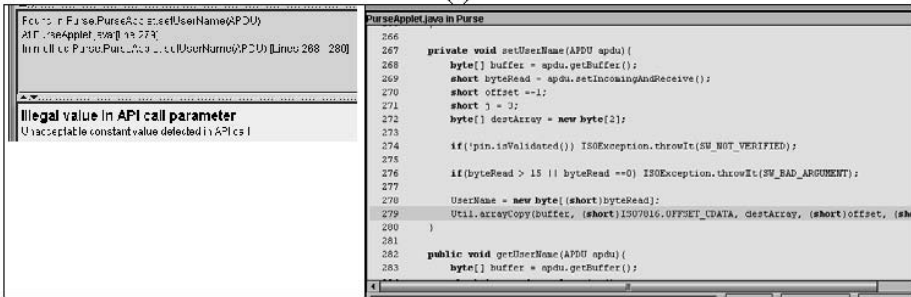
for each method in the class do
  request a CFG for the method
  get the method's ConstantDataflow from ClassContext
  get the method's ValueNumberDataflow from ClassContext
  get the method's IsNullValueDataflow from ClassContext
  for each location in the method do
    get instruction handle from location
    get instruction from instruction handle
    if instruction is not instance of invoke static then
      continue
    end if
    get the invoked method's name from instruction
    get the invoked method's signature from instruction
    if invoked method is arrayCopy then
      get ConstantFrame (fact) at current location
      get ValueNumberFrame (fact) at current location
      get IsNullValueFrame (fact) at current location
      get the method's number of arguments
      for each argument do
        get argument as Constant, ValueNumber, IsNullValue
        if argument is constant then
          if argument is negative then
            report a bug
          end if
        else
          if argument is not method return value nor constant then
            if argument is not definitely not null then
              report a bug
            end if
          end if
        end if
      end for
    end if
  end for
end for

```

Figure 7 demonstrates how the detector responds in two different property violation cases. In the first case, *PurseApplet* calls `arrayCopy` with null value for the parameter `accountNumber`. It is also important to note that it is not possible to determine by static analysis the correctness of the method call for all of the mentioned criteria, because `buffer` gets its value at run time by the JCRE. However, a complete FindBugs bug detector could generate a warning for the absence of an appropriate exception handler. In the second test case, parameter `offset` is assigned an unacceptable value.



(a)



(b)

Fig. 7. Illegal use of `arrayCopy` detected with (a) null value parameter and (b) unacceptable constant value parameter

5 Precise and Scalable Analyses for the Static Verification of API Calls

The static analysis case studies of Section 4 point out the merits as well as some shortcomings of the FindBugs open source framework, for the static verification of Java Card API calls. Although there is only limited documentation for the framework design and architecture, the source code is easy to read and self-documented. FindBugs is a live open source project and we will soon have new developments on shortcomings, like for example the lack of context-sensitive interprocedural dataflow analysis. Appropriate bug detectors can be supplied by the Java Card technology providers. Thus, Java Card applet providers will be able to use FindBugs in their development process with limited cost. This possibility opens new perspectives for automatically verifying the absence of unhandled security critical exceptions, as well as prospects for the development of bug detectors for application-specific correctness properties.

The static analysis techniques shown in the two case studies can be combined in bug detectors where either

- temporal safety includes constraints on the data values that are visible at the API boundary or
- we are interested to implement sophisticated and precise analyses that reduce false positives and at the same time scale to real Java Card programs.

In the following paragraphs we review the latest developments in related bibliography that address the second aim and in effect designate static program analysis as a credible approach for the static verification of security critical applications.

A notable success story in temporal safety checking is the ESP tool for the static verification of C programs. ESP utilizes a successful heuristic called “*property simulation*” [14] and a path feasibility analysis called “*path simulation*” [15], in order to perform partial program verification based only on the control-flow branches that are relevant to the checked property. This results in a selective path-sensitive analysis that maintains precision only for those branches that appear to affect the property to be checked. For one particular instantiation of the approach in which the domain of execution states is chosen to be the constant propagation lattice the analysis executes in polynomial time and scales without problems in large C programs like the GNU C compiler with 140000 LOC.

It is still possible to construct programs for which property simulation generates false positives, but the authors claim that this happens only to a narrow class of programs that is described in their article. Property simulation is designed to match the behavior of a careful programmer. In order to avoid programming errors programmers maintain an implicit correlation between a given property state and the execution states under which the property state machine is in that state. Property simulation makes this correlation explicit as follows:

- For a given temporal safety property, ESP performs a first analysis pass where it instruments the source program with the state-changing events.
- For the second analysis pass, the property simulation algorithm implements a merge heuristic according to which if two execution states correspond to the same property state they are merged. In any other case, ESP explores the two paths independently as in a full path-sensitive analysis.

Interprocedural property simulation requires generation of context-sensitive function summaries, where context sensitivity is restricted to the property states. This happens in order to exclude the possibility of a non-terminated computation that exists if the domain of execution states is infinite (e.g. constant propagation). Thus, execution states are treated in a context-insensitive manner: at function entry nodes, all execution states from the different call sites are merged.

The proposed path simulation technique manages execution states and in effect acts as a theorem prover to answer queries about path feasibility. In general, path feasibility analysis is undecidable. To guarantee convergence and efficiency, ESP makes conservative assumptions when necessary. While such over approximation is sound (i.e. does not produce false negatives), it may introduce imprecision. More recent research efforts in cutting down spurious errors that are at the same time scalable enough for solving real world problems focus on applying iterative refinement to path-sensitive dataflow analysis [16].

Another notable success story in temporal safety checking is the SAFE project [17] at the IBM Research Labs. Both ESP and SAFE build on the theoretical underpinning of a *typestate* as a refinement of the concept of type [18]. Whereas the type of a data object determines the set of operations ever permitted on the object, *typestate* determines the subset of these operations which are performed in a particular context.

Typestate tracking aims to statically detect syntactically legal but semantically undefined execution sequences. The heuristics applied in SAFE are reported in [19]. In that work the authors propose a composite verifier built out of several composable verifiers of increasing precision and cost. In this setting, the composite verifier stages analyses in order to improve efficiency without compromising precision. The early stages use the faster verifiers to reduce the workload for later, more precise, stages. Prior to any path-sensitive analysis, the first stage prunes the verification scope using an extremely efficient path-insensitive error path feasibility check.

The most serious restriction in the current version of FindBugs regarding the perspectives to implement sophisticated analyses like those described is the lack of support for interprocedural context-sensitive dataflow analysis. However, we expect that this restriction will soon be removed.

6 Conclusion

This work explored the adequacy of static program analysis for the automatic verification of Java Card applets. We utilized the FindBugs open source framework in developing two bug detectors that check the absence of unhandled security critical exceptions, concerned with temporal safety and correctness of the arguments of Java Card API calls. The developed detectors are sound, but they are not precise. We explored the latest developments that open new prospects for improving the precision of static analysis, thus making it a credible approach for the automatic verification of security critical applications. The results of our work and the bug detectors source code are publicly available online <http://mathind.csd.auth.gr/smart/>

A future research goal is the static verification of multi-applet Java Card applications (like the one in our case studies), in terms of temporal restrictions of inter-applet communications through shareable interfaces [20]. Also, we will continue to seek ways to overcome the experienced shortcomings in the current FindBugs version.

Acknowledgments

This work was supported by the funds of the bilateral research programme between Greece and Cyprus, Greek General Research Secretariat, 2006-2008.

References

1. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: a developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, Springer, Heidelberg (2003)
2. Beckert, B., Mostowski, W.: A program logic for handling Java Card's transaction mechanism. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 246–260. Springer, Heidelberg (2003)
3. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming* 58(1-2), 89–106 (2004)

4. Meyer, J., Poetsch-Heffter, A.: An architecture for interactive program provers. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 63–77. Springer, Heidelberg (2000)
5. Jacobs, B., Marche, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 241–257. Springer, Heidelberg (2004)
6. Van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
7. Breunese, C.B., Catano, N., Huisman, M., Jacobs, B.: Formal methods for smart cards: an experience report. *Science of Computer Programming* 55, 53–80 (2005)
8. The Java Verifier project,
http://www.inria.fr/actualites/inedit/inedit36_partb.en.html
9. Catano, N., Huisman, M.: Formal specification and static checking of Gemplus’s electronic purse using ESC/Java. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 272–289. Springer, Heidelberg (2002)
10. Meijer, H., Poll, E.: Towards a full formal specification of the JavaCard API. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 165–178. Springer, Heidelberg (2001)
11. Spinellis, D., Louridas, P.: A framework for the static verification of API calls. *Journal of Systems and Software* 80(7), 1156–1168 (2007)
12. The FindBugs project (last access: February 21, 2008),
<http://findbugs.sourceforge.net/>
13. Hovemeyer, D., Pugh, W.: Finding bugs is easy. *SIGPLAN Notices* 39(12), 92–106 (2004)
14. Dahm, M.: Byte code engineering with the BCEL API. Technical Report B-17-98, Freie University of Berlin, Institute of Informatics (2001)
15. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation (PLDI), pp. 57–68 (2002)
16. Hampapuram, H., Yang, Y., Das, M.: Symbolic path simulation in path-sensitive dataflow analysis. In: Proc. of 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), pp. 52–58 (2005)
17. Dhurjati, D., Das, M., Yang, Y.: Path-sensitive dataflow analysis with iterative refinemet. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 425–442. Springer, Heidelberg (2006)
18. The SAFE (Scalable And Flexible Error detection) project (last access: 21st of February 2008),
<http://www.research.ibm.com/safe/>
19. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering* 12(1), 157–171 (1986)
20. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA), pp. 133–144 (2006)
21. Chugunov, G., Fredlund, L.-A., Gurov, D.: Model checking of multi-applet Java Card Applications. In: Proc. of the 5th Smart Card Research and Advanced Application Conf. (CARDIS) (2002)

On Practical Information Flow Policies for Java-Enabled Multiapplication Smart Cards*

Dorina Ghindici and Isabelle Simplot-Ryl

IRCICA/LIFL, CNRS UMR 8022, Univ. Lille 1, INRIA Lille - Nord Europe, France
{dorina.ghindici,isabelle.ryl}@lifl.fr

Abstract. In the multiapplicative context of smart cards, a strict control of underlying information flow between applications is highly desired. In this paper we propose a model to improve information flow usability in such systems by limiting the overhead for adding information flow security to a Java Virtual Machine. We define a domain specific language for defining security policies describing the allowed information flow inside the card. The applications are certified at loading time with respect to information flow security policies. We illustrate our approach on the LoyaltyCard, a multiapplicative smart card involving four loyalty applications sharing fidelity points.

1 Introduction

Computer systems handle a considerable amount of data carrying sensitive information that should be protected from malicious users. Programs running on such systems may access data either to perform computations or to transmit it over an output channel. Thus they can violate the security of sensitive data either by releasing it to unauthorized users or by modifying it. In order to prevent such situations, tracing data manipulation throughout programs is mandatory.

Information flow analysis [16] consists in statically analyzing the code of a program in order to detect illicit data manipulations. Concretely, data manipulated by programs (e.g. objects, parameters) are tagged with security labels and all information flows are traced. The assignment $p := s$, where p is a public, observable variable and s contains secret, confidential data, generates an *explicit* flow from secret to public data. The code $\text{if}(s) \ p := 0$ contains an *implicit* flow of information as an external observer, who has knowledge about the control flow of the program, can learn information about the secret data s . Usually, information flow is associated with non-interference [9] which prevents all information flows from sensitive data to non-sensitive data. The examples above generate illegal information flows w.r.t non-interference.

Information flow analysis does not guarantee security by itself: it is a powerful mechanism that can be exploited to implement the desired security policies. The difficulty is to ensure that local checks (mechanisms) actually implement the global security policy. Information flow mechanisms are too coarse to express

* Funded by ANR SESUR SFINCS (ANR-07-SESU-012) project.

desired policy, thus one of their common pitfalls is to define and verify complex policies, reflecting real attacking scenarios.

In this paper, we address the problems of defining security policies for information flow, enforcing them by an information flow analyser and helping the programmer to build safe applications in case the verification fails. The target devices are multiapplicative smart cards, running a Java Virtual Machine (JVM). The security policies express allowed data flow between applications, either due to code reuse or collaborations (e.g. commercial agreement). To support our approach, we consider the case study of LoyaltyCard, a multiapplicative smart card containing four fidelity applets. The main contributions of this paper are:

- to define a specific language for specifying information flow security policies,
- to present how the policies are enforced in a standard JVM,
- to make the information flow analysis practical w.r.t. software engineering, by adding information flow contracts and giving hints for helping programmers to develop safe applications.

The rest of the paper is structured as follows: Section 2 presents the LoyaltyCard example, while Section 3 introduces some aspects of information flow analysis in the context of open, small systems and identifies challenges. In Section 4 we define a domain specific language for information flow policies, while Section 5 presents a deployment and development environment. Section 6 discusses related work, while Section 7 summarizes our contributions.

2 LoyaltyCard Example

In this section we present LoyaltyCard, a multiapplicative Java-enabled smart card composed of four loyalty applications: two air companies (FlyFrance, FlyMaroc), a car renting company (MHZ) and a hotel (Illtone). The applications implement loyalty services and can share information. Three of these applications form a group of partners: confidential data flow between partners is secure, while collaborations with external applications, as depicted in Figure 1, may lead to illegal flows of information.

Let us suppose that FlyFrance has a commercial agreement with MHZ and Illtone, so part of FlyFrance points can be used to obtain MHZ and Illtone loyalty points. On the other hand, FlyFrance does not want FlyMaroc to learn any information about the fidelity status of its clients (e.g. the number of miles, or the status: gold, silver client, etc). FlyMaroc has also an agreement with MHZ and offers a discount, based on the fidelity status of the MHZ client. Suppose that, when asked by FlyMaroc, MHZ returns not only its fidelity points, but also fidelity points of its partners (FlyFrance). FlyMaroc can infer, through MHZ, information about the FlyFrance fidelity points, as depicted in Figure 1. In such

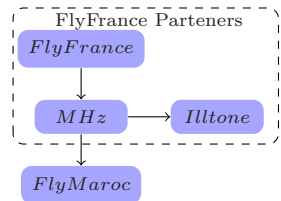


Fig. 1. Illegal information flow

```

class FlyFrance {
    private int miles;
    [...]
    public void updates() {
        int i=0;
        for(;i<noLoyalties;i++)
            update(loyalties[i]);
    }
    void update(Loyalty l){
        l.update(miles);
    }
}

class MHz extends Loyalty{
    private int points;
    private int ppoints;
    [...]
    public void update(int p){
        this.ppoints += p;
    }
    public int getLevel_() {
        if(points+ppoints>GOLD)
            return LEVEL_GOLD;
        return LEVEL_SILVER;
    }
}

class FlyMaroc {
    private int oldLevelMHz;
    [...]
    int makeGetLevel(MHz h) {
        int newLevel = h.getLevel();
        if(oldLevelMHz!=newLevel){
            print("level changed!");
            oldLevelMHz = newLevel;
            return newLevel;
        }
        return ERROR;
    }
}

```

Fig. 2. Excerpt from the Java implementation of LoyaltyCard

a way, an illegal information flow is established. Illtone also offers a discount for MHz clients, but this time the flow of information is allowed, as Illtone is one of the partners of FlyFrance.

We want to be able to show that the implementations of the FlyFrance, MHz, FlyMaroc and Illtone enforce information flow policies, e.g. each program shares data only with trusted applications.

Figure 2 shows an extract of the Java code of FlyFrance, FlyMaroc and MHz classes. The confidential data of FlyFrance is stored in the field `miles`. The method `update` in FlyFrance updates the points of its partners (MHz and Illtone). MHz stores its partner points in field `ppoints`. Method `MHz.getLevel()` returns the fidelity level of MHz, based on MHz points and on partner points. This method is called by FlyMaroc in order to offer a discount, which leads to an unexpected flow of information to an application untrusted by FlyFrance. The `MHz.getLevel()` method is also called by Illtone, but in this case, the flow of information is authorized as FlyFrance has an agreement with Illtone.

3 Embedded Security and Information Flow

Ubiquitous computing is evolving towards post issuance and automatic execution of untrusted code. Executing untrusted code implies many security risks. For example, a malicious applet running on your mobile phone or smart card can do a lot of harm: it can disclose confidential information, financial data, address book, social security and medical files, etc. Moreover, if the system runs multiple applications, which share data, then it must ensure data confidentiality for each application by controlling the underlying information flow.

3.1 Information Flow Model

In [6], a compositional information flow analysis enforcing non-interference for Java programs running on small, open embedded systems has been presented. The analysis consists in statically interpreting JVM bytecode and inferring types

representing all possible information flows that may occur when executing the program. The behaviour of a program, in terms of information flow, is defined using contracts: a security contract [1] guarantees the maximal information flow that may occur while executing the program. In this paper, we enrich this framework with security policies which relax non-interference by describing allowed flows of information between application.

We now briefly describe the information flow model, which is needed for better understanding the identified challenges and the proposed solutions. For more details on the model, please refer to [6].

Information Sources and Security Levels. As in classical information flow, each field is annotated by a security level: s for secret, sensitive data and p for public, observable data. This work is based on the idea that confidential data in small objects (e.g. loyalty points, PIN code) typically resides in instance fields of objects [10]. We prevent information flow from high-security to low-security instance fields. In order to have a reasonable size of information flow annotations for embedded systems, the analysis is *field independent* but *security level sensitive*: all the fields of an object having the same security level are modeled as having the same location. Thus, considering the security levels s and p , each object o is modeled as two sub-objects (parts): a secret part (o^s) and a public part (o^p).

In the LoyaltyCard example, the confidential data (FlyFrance fidelity points), is stored in the field `FlyFrance.miles` and the field `ppoints` of its partners (`MHz` and `Illtone`). Hence these fields have security level s .

The Flow Relation. We now define the flow relation. We say that there is a flow from a to b if an observer of b can learn information about a .

Considering our split of objects and the dichotomy of Java types (elementary types and object types), the flows between two elements a and b have the form $a^{\wp(p,s)} \xrightarrow{\mathbf{r}/\mathbf{v}/\mathbf{i}} b^{\wp(p,s)}$, where \mathbf{v} denotes a flow arising from an assignment of primitive type, \mathbf{r} a reference flow (an alias), \mathbf{i} an implicit flow; s and p denote the security levels, secret or public, while $\wp(p,s)$ denotes subsets of $\{p,s\}$.

For example, if an object \mathbf{a} has a field \mathbf{s} , of type `int`, labeled with security level s and \mathbf{b} has field \mathbf{p} with security level p , the code `$\mathbf{b.p} = \mathbf{a.s}$` generates a value flow from the secret part of \mathbf{a} to the public part of \mathbf{b} , denoted by $b^p \xrightarrow{\mathbf{v}} a^s$. The code `$\text{if}(\mathbf{a.s}) \mathbf{b.p}=0$` generates an implicit flow $b^p \xrightarrow{\mathbf{i}} a^s$.

The Security Contract of a Method. A security contract [19] carries relevant information for a later usage of the method: it contains flows, potentially generated by the execution of the method, between sources of information flow (abstract values) visible outside the method. We identify thus the following abstract values:

- the parameters of a method m ,
- the return value of the method, denoted by the abstract value R ,
- input/output channels: all the channels are abstracted by a single value, IO ,

- static fields, which are modeled as fields of a single object, denoted by the abstract value *Static*,
- exceptions: all thrown values flow to the abstract value *Ex*.

Let Σ_m be the set of abstract values of a method m . We define now the security contract of a method m as

$$S_m = \{a \xrightarrow{\mathbf{r}/\mathbf{v}/\mathbf{i}} b \mid a, b \in \Sigma_m \times \wp(p, s) \text{ and the execution of } m \text{ potentially generates a flow from } a \text{ to } b\}.$$

For example, considering that `FlyFrance.miles` and `MHz.ppoints` have security level s , the method `MHz.update` from the `LoyaltyCard` (Figure 2) generates a flow from the parameter \mathbf{p} to a secret field of `this`. Thus, the security contract of the method is $S_{update} = \{this^s \xrightarrow{\mathbf{v}} p\}$. The information flow analysis infers the security contracts in Figure 3 for the rest of the methods in Figure 2.

<i>FlyFrance</i> : S_{update}	$= \{l^s \xrightarrow{\mathbf{v}} this^s\}$
<i>MHz</i> : S_{update}	$= \{this^s \xrightarrow{\mathbf{v}} p\}$
<i>MHz</i> : $S_{getLevel}$	$= \{R \xrightarrow{\mathbf{i}} this^{s,p}, R \xrightarrow{\mathbf{v}} Static\}$
<i>FlyMaroc</i> : $S_{makeGetLevel}$	$= \{this^p \xrightarrow{\mathbf{i}} h^{s,p}, R \xrightarrow{\mathbf{i}} h^{s,p}, R \xrightarrow{\mathbf{v}} Static\}$

Fig. 3. Security contracts for LoyaltyCard

3.2 Challenges

The real challenge in information flow analysis is applying its results in practice. We identify some of the major problems in making the analysis usable for which we give solutions in the next sections.

Defining Policies that Explore Security Contracts. In literature, confidentiality is often seen as a non-interference [9] problem, as public outputs cannot depend on secret inputs. Non-interference policies do not allow any flows from secret to public values, but only flows from secret to secret. Nevertheless, non-interference does not make any distinction between the source of secrets. It is a transitive and symmetric relation. Policies defined with such relation are too restrictive, and not the desired policies in most of the cases, and especially in multiapplicative smart cards [8]. Our aim is to refine non-interference by defining more complex intransitive and asymmetric policies. In the `LoyaltyCard`, the security policies of an applet running on the smart card (`FlyFrance`) allow secret information to be released to some other applet (`MHz`) but not to `FlyMaroc`.

In order to escape from non-interference strictness, we define, in Section 4, a specific language, which describes the allowed flow of information between applications. Programs are certified by verifying that the security contracts respect the desired security policies.

Integration to Existing Systems (Jvm). Another important challenge, which prevented information flow mechanisms from being used in real systems, is getting the certification process and information flow policies to correctly and easily interact with existing systems. JFlow [13] and Flow Caml [18] are powerful languages, that offer support to a reliable development by defining a new programming language which mixes source code and security policies in a coherent set. However, they do not address the problems raised by mobile code and open environments, and do not fit into the Java paradigm of dynamic class loading.

Integrating information flow policies for mobile code in Java-enabled small open embedded systems requires, at least, (i) separation of code and security policies and (ii) certification at load time. In Section 5.1 we present how the enforcement process of information flow policies is integrated in a JVM.

Developing Safe Interacting Applications. The security of a system depends on the security of each component. A key in computer security is not only detecting and preventing attacks, but also helping the developer to build safe applications, components. Contracts can be used as a support for creating secure applications, as developers can express, by their means, the expected behaviour of unknown or untrusted applications. In this paper, we express information flow in a program using contracts and we introduce an approach, based on reverse engineering, to help building applications that respect information flow policies.

4 A DSL for Information Flow Policies

A domain-specific language (DSL) is a small, usually declarative, language that targets a particular kind of problem. The key characteristic of DSLs is their focused expressive power. DSLs are usually concise, offering only a restricted suite of notations and abstractions, thus adapted to express security policies.

4.1 DSL Definition

In order to express security policies describing collaborations and information flows between applications, we thus define a domain-specific language in Figure 4. The security policies that can be expressed with the DSL are simple, but have enough power to model collaborations schemes in a smart card. The DSL was designed for multiapplication smart cards, but it can be extended to other applications, if necessary.

Multiapplicative smart cards allow data sharing and service sharing in order to optimize the use of resources (e.g. API) and to allow collaborative schemes (e.g. agreements or contracts between applications). In a smart card, the entities exchanging or sharing data are the applications, thus the DSL contains rules defining *trust relations* between applications.

Applications are addressed either by package or class names, using elements in the sets of terminals *Class* and *Package*; *Field* denotes a set of terminals containing field names.

As we consider that confidential data resides in class fields, rule R_c expresses the secrets of a class, by listing the fields that should remain confidential, and thus that have the security level s . The main rule of the DSL is R_s which describes the allowed information flows. For example, the signification of S_1 **shares with** S_2 ; is that all elements in S_1 can share their secrets with all elements in S_2 .

By default, an element in S can share its secret with other elements having the same type (e.g. class A shares its secrets with all instances of class A). Rule $R_p = S$ **strict secret** ; refines the security policies by specifying that an element A in S must not share its secrets with other objects of type A . While the rule R_s defines type-based policies, rule R_p refers to instance-based policies, in the case when the instances have the same type.

The sharing relation can be associated to the *trust relation* defined in [8] by Girard: one application transmits its secrets only to trusted applications. As the trust relation, the sharing relation is neither symmetric nor transitive. If A **shares with** B then not necessarily B **shares with** A . An application would not trust another application only because one of its trusted applications does. Detecting data leaks due to transitivity, or propagation, is one of the main concerns of information flow security. Allowing transitivity would make no distinction between information flow and access control.

4.2 DSL Verification

The certification process of an information flow policy has two parts:

1. verifying *simple class sharing*: an application gives its secrets only to trusted applications,
2. verifying *transitivity* (or data propagation): an application trusted by A does not share confidential data with applications untrusted by A .

Simple Class Sharing. The rule P_i **shares with** P_j , where P_i and P_j are two packages, can be read as "any class in package P_i trusts any class in package P_j ". Hence, the DSL in Figure 4 can be reduced to rules having the form A **shares with** B , where A and B are class names in $Class$. We can compute a function $share : Class \rightarrow \wp(Class)$ which associates to each class the classes it trusts. By default, a class trusts itself, thus A **shares with** A ; and $A \in share(A)$. Verifying the security policy of a class $A \in Class$ reduces to verifying that secrets of A flow only to elements in $share(A)$. Hence, we verify security policies at the granularity level of classes.

Transitivity. Once a class A shares confidential data with a trusted class B , A loses control over its propagation. The secret of A becomes the secret of B .

```

S ::= (Class|Package)[,S]
F ::= Field[,F]
R_c ::= Class secret F;
R_s ::= S shares with S;
R_p ::= S strict secret ;
P ::= (R_c|R_s|R_n)[,P]

```

Fig. 4. A DSL for information flow policies

The policy of A holds if the policy of B is more restrictive: B does not share its secrets with applications untrusted by A . Formally, verifying transitivity can be summed up to verifying that, for all $B \in \text{share}(A)$, $\text{share}(B) \subseteq \text{share}(A)$ holds.

Security Policies and Security Contracts. We now show how policies defined using the DSL are enforced by the information flow analysis described in Section 3. Confidential data resides in object fields. Let $\text{fields}_s : \text{Class} \rightarrow \wp(\text{Field})$ be a function that associates to each class the fields having the security level s , thus fields in the rule $R_c = \text{Class } \mathbf{secret} \text{ Field}$; the function $\text{fields} : \text{Class} \rightarrow \wp(\text{Field})$ gives all fields of a class.

Secrets can be made accessible either by direct access to fields or through method invocations and operations performed by the method. In order to prevent direct access, secret fields of a class A in $\text{fields}_s(A)$ must be declared using the Java access modifier *private*. This restricts the access to secret fields only in the class where they have been declared and thus to which they belong. Based on this, certifying a class A with respect to an information flow policy consists of verifying every method in A and methods that use the class A . Let Method be the set of method names and $\text{methods} : \text{Class} \rightarrow \wp(\text{Method})$ a function that gives the list of methods for each class.

Let us remind that the information flow model in Section 3 computes, for each method $m \in \text{Method}$, a security contract S_m containing all the possible flow of information between abstract values in Σ_m (parameters, *IO*, *Ex*, *Static*, *R*). A flow is denoted by $a^{t_1} \xrightarrow{\mathbf{r}/\mathbf{v}/\mathbf{i}} b^{t_2}$ with $t_1, t_2 \in \{s, p\}$ denoting the security level. Flows can be from public/secret parts of an abstract value to public/secret parts of another abstract value. Security is concerned with protecting flows from the secret parts to public/secret parts. As a general rule, flows from secret to public are forbidden, while flows from public to public are always allowed. A class A shares its secrets with classes in $\text{share}(A)$, thus only flows from secret parts of parameters of type A to parameters with type in $\text{share}(A)$ and to return (*R*) are allowed. Let T be a function which associates to an abstract value its definition type, in Class . The algorithm that verifies method in a class A is depicted in Figure 5. To permit the flows to return, we consider that $R \in \text{share}(A)$.

```

1: for all m in methods(A) do
2:   if  $\exists a^p \xrightarrow{f} b^s \in S_m$  then
3:     return false
4:   end if
5:   for all  $a^s \xrightarrow{f} b^s \in S_m$  do
6:      $t_1 = T(a)$ ,  $t_2 = T(b)$ 
7:     if  $t_1 \notin \text{share}(t_2)$  then
8:       return false
9:     end if
10:  end for
11: end for
12: return true

```

Fig. 5. Certifying the policy of class A

Encapsulation. The split of objects and the definition of secret/public part may open a door to bypassing security checks through encapsulation. For example the code $A.p.r=A.s$, where s and r have security level s and $T(p) \notin \text{share}(A)$, generates a flow $A^s \xrightarrow{\mathbf{v}} A^s$, allowed by our analysis, but illegal as the secret of A flows to an untrusted type (p). In order to avoid such leaks, we define the fol-

lowing encapsulation property: *All secret fields and sub-fields of a class A must be trusted by A*, where sub-fields refer to fields of fields and etc.

The verification of this property consists in unfolding the fields of each class and verifying that, for each secret field f , we have $\mathcal{T}(f) \in \text{share}(A)$. Let $\text{enc}_{\text{field}}(A, B)$ be a function which verifies, recursively, that all secret fields of class A are trusted by B ($\text{enc}_{\text{field}} : \text{Class} \times \text{Class} \rightarrow \{\text{true}, \text{false}\}$). The algorithm is depicted in Figure 6.

If we take in consideration that only few classes contain secret fields, we can label the classes containing only public fields and stop the unfolding when we meet such classes. Let $\text{scr}_C : \text{Class} \rightarrow \{\text{true}, \text{false}\}$ be a function which tests if a class contains some secret fields or not; $\text{scr}_C(A)$ refers not only to secrets defined in A but also to secrets defined in fields of A , etc.

Thus, to verify that a class A respects the encapsulation property, a call to $\text{enc}_{\text{field}}(A, A)$ is sufficient.

4.3 Example

Figure 7 presents the information flow policy for the LoyaltyCard presented in Section 2. The first three rules define the confidential data, while the last two rules define the allowed information flow. The policy respects transitivity, as the policies of applications trusted by FlyFrance (MHZ, Illtone) are smaller than the policy of FlyFrance. The verification fails while trying to validate the method `makeGetLevel` defined in `FlyMaroc`,

as it contains a flow $\text{this}^p \xrightarrow{i} h^s$, where h denotes the MHZ application.

4.4 Discussion

Conflict Resolution. While rule R_c and R_s are permissive, the rule R_p is restrictive and thus can generate conflicts. Let us consider the following policy for a class $x.A$, where x is the package to which A belongs:

$x.A$ **strict secret** ; $x.A$ **shares with** $x.*$;

```

for all  $f$  in  $\text{fields}_s(A)$  do
  if  $\mathcal{T}(f) \notin \text{share}(B)$  then
    return false
  end if
end for
for all  $f$  in  $\text{fields}(A) \setminus \text{fields}_s(A)$ 
do
  if  $\text{scr}_C(\mathcal{T}(f))$  then
    return  $\text{enc}_{\text{field}}(\mathcal{T}(f), B)$ 
  end if
end for
return true

```

Fig. 6. $\text{enc}_{\text{field}}(A, B)$

```

FlyFrance secret miles;
MHz secret ppoints;
Illtone secret ppoints;
FlyFrance shares with
MHz, Loyalty, Illtone;
MHz shares with Illtone;

```

Fig. 7. Security policy for Loyalty-Card

In the first rule, $x.A$ does not trust itself, while in the second rule $x.A$ trusts all classes in package x , and thus it trusts itself. To solve such conflicts, we consider that the rules R_p (**strict secret**) prevail over rules R_s (**shares with**). Thus, we first construct the function *share*, and only after we take into consideration the fact that a class is **strict secret** or not.

Support for Overloading. One of the most powerful attributes of object-oriented programming, and thus Java, is code reuse and factorisation, by the means of inheritance. But, apart from providing this powerful functionality, inheritance provides also means for leaking information. To prevent such leaks, we define some relations between policies of subclasses and superclasses.

The first restriction regards inherited fields: their security level cannot be changed by a subclass. Doing so, the security contracts of inherited methods change, and the superclass must be reanalyzed. This is not convenient for our compositional approach, and for open systems. Nevertheless, a child class can declare new fields even with security level s .

While overloading a class, for example B extends A , the security policy of B must not only enforce security for B , but also for A and classes already verified using A . If the policy of B is greater than the policy of A , formally $share(B) \supseteq share(A)$, then the confidentiality of A is not respected anymore, as B can trust and share its secrets (and thus those of A) with classes which A does not trust. If the policy of B is smaller than the policy of A , formally $share(B) \subseteq share(A)$, in order to certify B we must reanalyse A , as A , and thus a part of B , have been certified using a greater policy. From these examples, we can conclude with: the security policy of a class B must be the same as the security policy of its superclass A , $share(B) = share(A)$.

The constraint above is too strict for API classes, which are *public classes* (we use this term to denote classes which do not contain secrets, hence classes for which scr_C returns *false*). In order to deal with API, we relax the policy above in the following way: the policy of a subclass must be the same as the policy of the inherited class only if the inherited class contains secret fields. Thus, the policy of a subclass can be any policy, if the inherited class is a public class. Problems may arise if we cast a public class to a class which contains secrets. To deal with such situations, we extend the flow signature with the types in which public classes are cast inside the method, and we take into consideration all these types while verifying the method.

For example, let us consider that we have C extends B . There are 2 cases: Security issues arise when class B does not contain any secrets ($fields_s B = \emptyset$) and C declares secret fields ($fields_s(C) \neq \emptyset$). In this case, the leak occurs only when a cast from B to C is made inside a method m . To solve this problem, while

```

1: if  $fields_s(B) \setminus fields_s(A) \cap$ 
    $fields(A) \neq \emptyset$  then
2:   return false
3: end if
4: if  $fields_s(A) \neq \emptyset \wedge$ 
    $share(B) \neq share(A)$  then
5:   return false
6: end if
7: return true

```

Fig. 8. Certifying the policy of class B extends A

analysing m , we store the types in which classes of type B are cast inside m ; for example, if parameter p_1 of type B is cast in C or in D , then we associate a list to p_1 ($p_1 \Rightarrow (C, D)$). This list must be kept only for types which do not contain secret fields, thus for which the function scr_C does not hold. Flow signatures are extended with such lists. For simplicity, we do not consider this case in the algorithm presented below.

We can now extend the certification algorithm presented in Figure 5 to take into consideration overloading. The extension is presented in Figure 8. Readers should not confuse security policies with security contracts, for which we have different restrictions.

Extending Policies (Declassification). The DSL and the security policies can be extended to express more detailed rules about the release of information. The current DSL expresses policies that apply to entire program, and does not specify *where* the information release is permitted. We can define rules that delimit the methods where the information flow may occur, for example

$$R_m ::= S \text{ shares with } (\mathbf{IO} \mid S) \text{ in } Method;$$

where *Method* represents a method name or a list of methods and **IO** is a keyword (terminal) standing for the abstract value *IO*. The declassification adds power of expression as it allows also to send data on input/output channels.

Declassification relaxes the security policies in certain method. To support polymorphism and dynamic class loading, all the overriding classes must agree on the declassification contract, e.g. the declassification rule must be defined by every class in the class hierarchy.

Information Flow Policies as Contracts. The DSL in Figure 4 allows the declarations of information flow policies for applications sharing confidential data. Not only this language has a declarative value, but it also has a contractual value. For example, with the rule *FlyFrance shares with MHz*, FlyFrance imposes a contract to MHz: FlyFrance agrees to share its secrets with MHz only if MHz does not share its secrets with applications not trusted by FlyFrance. Thus, the policies defined using the DSL are contracts that applications must respect. An application accepts the contract of a trusted application only if it is smaller than its own contract. In order to deal with openness and overriding, the DSL imposes that the contracts of classes extending classes containing confidential data do not change, with respect to the contract of overridden class.

5 Integrating Information Flow in a Development and Deployment Schema

Even if information flow is a well studied area, there are not enough mechanisms guaranteeing security for existing systems. The main difficulty for practical information flow is to integrate it in a real development and deployment schema.

5.1 Enforcing Security Policies for Jvm

We present here how information flow policies defined in previous section may be enforced by any JVM. As the compiled JVM bytecode is downloaded through an unsecured channel, the information flow certification must be done oncard, preferably at loading time in order to avoid run-time overhead. Both security contracts and policies must be enforced. As computing security contracts requires many resources (both in memory and time), we perform a two step analysis: (i) an external phase [6] (supposed to have access to infinite resources) which computes the type inference and annotates the bytecode with some proof elements, and (ii) an embedded phase [7], which verifies, at loading time, the security contracts obtained during the external phase. The verification operation is linear in code size and uses constant memory. This technique lies on the same simple idea as proof-carrying code [15] that it is easier to verify a result already computed. We deal here only with the verification of information flow policies. The verification of security contracts is described in [7].

In order to make the analysis practical and integrable with any existing JVM system, we (i) load policies to be certified as attributes of .class files; systems not enforcing information flow can ignore these attributes, and (ii) verify security policies with a custom class loader, that can be installed on any system.

Extending .class Files with Information Flow Security Policies. The policy of a class A is the list of classes with which it can share confidential data (denoted by $share(A)$). The .class attribute for the policy of A contains thus a list of class names. The class names are represented by their index in the ConstantPool of the class A . Considering that in a smart card the number of installed applications is not significant, thus the sharing policies are quite simple, the newly added attribute contains usually only few entries. The small size of the attribute is acceptable for a small system.

As classes are loaded one by one, it is possible to load A before loading all the classes used by A . While validating a class A , we also validate the policies of classes used in A . Thus, to be able to validate A , we also load the policies of classes used in A . If B is a class used by A , when loading A either (i) we take in consideration the policy of B , if B has already been loaded or (ii) use the policy of B that A announces and we keep it oncard, in a repository, in order to validate (and remove) it when B is loaded.

Verifying Security Policies Using a Custom Class Loader. The loading process in a JVM is performed by the class loaders. In order to integrate the information flow analysis on any JVM, the verification is performed by custom a class loader (*SafeClassLoader*), which can be built in the single class loader of KVM or installed as a user-defined class loader for a standard JVM. The *SafeClassLoader* must verify both security contracts, as described in [7], and information flow policies.

Classes are loaded one by one. Once the security contracts of the class have been verified, the *SafeClassLoader* validates the information flow policy, using the security contracts. The difficulty may arise from the fact that the loaded

class A wants to share its secret with a class B not yet loaded. As the class is not present in the system, we do not have its security policy and we cannot verify the transitivity, formally $share(B) \subseteq share(A)$. In order to verify this condition when B is loaded, we keep a repository with rules having the form $share(B) \subseteq share(A)$. If B is used by another class C , the rule $share(B) \subseteq share(C)$ must be added to repository. In this case, the final rule kept in the repository is $share(B) \subseteq share(A) \cap share(C)$, as the policy of B should be more restrictive than both policies of A and C . Thus, when the load B , we also verify that $share(B) \subseteq X$ with X denoting the intersection of security policies of classes that trust B . Moreover, we verify that the loaded class has the same policy as its super class: $share(B) = share(B')$ with B extends B' .

Verifying Encapsulation. The same problem may arise when verifying encapsulation: A has a field of type B , but B is not yet loaded. In order to verify while loading B that all secret fields of B are trusted by A , we keep the following rule to the repository: $enc_{field}(B, A)$. When loading B , if a rule $enc_{field}(B, A)$ is found in the repository, than the function $enc_{field}(B, A)$ (see Figure 6) is executed. If the test succeeds, the rule is deleted from repository and the loading process continues, by performing other checks.

The result of $enc_{fields}(B, A)$ depends on $scr_C(B)$ (the function which tests if B or fields B contain secret fields). The value returned by $scr_C(B)$ depends also on fields of B . Hence, the final value of $scr_C(B)$ can be computed only when all fields, fields of fields, etc. have been loaded. To ensure the correctness of scr_C computation, we extend the repository with rules of type $scr_C(B) = scr_C(C_1) \vee scr_C(C_2) \vee \dots \vee scr_C(C_n)$, where $C_1 \dots C_n$ represent the type of fields of B not yet loaded. This rule is deleted from repository when a class C_i is loaded with $scr_C(C_i) = true$ or when all classes $C_1 \dots C_n$ are loaded. Moreover, to avoid the computation of scr_C each time when it is needed, the known values of scr_C are stored on the card, in a special repository.

The algorithm verifying encapsulation at loading time is similar to the external one (enc_{field}) presented in Figure 6, except that it must also verify that the class $T(f)$ has been loaded; if not, it should add $enc_{field}(B, A)$ to the repository.

5.2 Reverse Engineering Tool

The certification must be done oncard due to the fact that the applications are loaded using an unsecured channel and must be adapted to the limited resources of the system. In the same time, the external analysis is supposed to be done on an system offering of unlimited resources comparing with a small system. Thus optimization and complexity are not an issue. Moreover, the external resources can be used for other purposes, for example for offering an easy development environment to programmers.

Security must be insured for different attacks against computing systems, for both deliberate or accidental attacks. Information flow insecurity may arise from malicious, untrusted code or from our own code. In the later case, the insecurity is due to bad conception of the application or to bad implementation. When

the information leak comes from a bad implementation due to human error, it is not always obvious for the developer to correct the application in order to make it safe. The development environment should detect illicit flows and help the developer to correct his mistakes by offering all the necessary information.

The point of failure in the program certification is not usually the real source of information leak. For example, the certification of `LoyaltyCard` fails while analysing the method `FlyMaroc.makeGetLevel`. But the illegal information flow comes from the implementation of method `getLevel` in class `MHZ`, where the computation of fidelity level for `MHZ` takes into consideration the points of partners.

To detect the failure source, we propose a backward iterative algorithm, which, at each step, tries to detect an information flow in a method. The algorithm is similar to tracking thrown exceptions in Java programs. Let us assume that we have a recursive method $detect(m, f, pc)$ which detects where the flow f occurred in method m by performing a backward analysis starting from the program point pc . If the flow f was created due to another flow f_1 , the method $detect(m, f_1, pc)$ is called recursively. If the flow f was created in a method m_1 invoked at pc , the algorithm calls $detect(m_1, f, pc_f)$, where pc_f is the program counter corresponding to the return instruction in method m_1 .

This approach is memory consuming and thus cannot be performed oncard, but it can explore the unlimited resources of the external analyser.

6 Related Work

Information flow [16] has been largely studied in the last decades and many models have been proposed [2,10]. Unfortunately, these models are mostly theoretical and almost impossible to apply in practice. Complex programming systems [13,18] enforcing information flow security exist, but they failed in showing how they can be successfully applied to real problems [20]. Most of the systems enforce standard non-interference and expressive, useful information flow security policies lack. The PACAP framework [4] involves a technique based on model checking to verify interactions for Java smart-cards, but the verification is limited to predefined scenarios, and it cannot be trusted in an open environment.

Several works have developed policies for downgrading data [17]. JFlow [13], a powerful programming language, implemented as an extension of the Java language, implements the decentralized label model (DLM) [14] which uses the notion of ownership; data can be release only by one of the owners only if all the owners agree. This approach is similar to our contracts on declassification: data can be released in a method if all classes in the hierarchy agree on the release. JFlow adds reliability to software implementation, but not to deployment and linking on a platform. Moreover, source programs must be annotated with security labels, and hence they must be re-coded. Many other forms and systems that declassify information have been presented [5,12] but most of them are certified by a security type system and are based on the assumption that policies are known statically at compile time. All these work have solid theoretical foundations, but failed to be successfully applied in practice.

On the other hand, many domain specific languages and practical systems expressing security policies exist [3,11], but they do not address information flow issues and most of the time they are dynamically enforced. Domain specific languages [11] limit themselves to specifying access control rules and do not address data propagation.

7 Conclusion

Motivated by the LoyaltyCard example, we present an approach to detect illegal information flows in multiapplicative smart cards. The desired security policies are specified using a simple, but expressive domain specific language and are enforced at loading time. On the one hand, this work bridges the gap between information flow models and current running systems. While the foundations of information flow models are solid, their practical side is still to be proved. Our approach limits the overhead for adding information flow security to existing JVM, as security labels and policies are separated from the code, and the illegal information flow is detected by a custom class loader, installed on any JVM. On the other hand, our work bridges the gap between information flow security requirements and actual security policies, which do not take into consideration data propagation due to information flow.

References

1. Aissa, N.B.H., Ghindici, D., Grimaud, G., Simplot-Ryl, I.: Contracts as a support to static analysis of open systems. In: Proc. of 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2007) (2007)
2. Avvenuti, M., Bernardeschi, C., Francesco, N.D.: Java bytecode verification for secure information flow. ACM SIGPLAN Notices 38(12), 20–27 (2003)
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Proc. ACM SIGPLAN Conf. PLDI 2005, pp. 305–314 (2005)
4. Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V., Zanon, G.: Checking secure interactions of smart card applets: extended version. J. Comput. Secur. 10(4), 369–398 (2002)
5. Dam, M., Giambiagi, P.: Confidentiality for mobile code: The case of a simple payment protocol. In: Proc. 13th IEEE CSFW 2000, pp. 233–244 (2000)
6. Ghindici, D., Grimaud, G., Simplot-Ryl, I.: Embedding verifiable information flow analysis. In: Proc. Conf. Privacy, Security and Trust (PST 2006), pp. 343–352 (2006)
7. Ghindici, D., Grimaud, G., Simplot-Ryl, I.: An information flow verifier for small embedded systems. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 189–201. Springer, Heidelberg (2007)
8. Girard, P.: Which security policy for multiapplication smart cards? In: USENIX Workshop on Smartcard Technology, pp. 21–28 (1999)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symp. Security and Privacy, pp. 11–20 (1982)
10. Hansen, R.R., Probst, C.W.: Non-interference and erasure policies for java card bytecode. In: 6th Intl. Workshop on Issues in the Theory of Security (WITS 2006) (2006)

11. Hashii, B., Malabarba, S., Pandey, R., Bishop, M.: Supporting reconfigurable security policies for mobile programs. *Comput. Networks* 33(1-6), 77–93 (2000)
12. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. *ACM SIGPLAN Notices* 40(1), 158–170 (2005)
13. Myers, A.C.: JFlow: practical mostly-static information flow control. In: *Proc. 26th ACM SIGPLAN-SIGACT Symp. POPL 1999*, pp. 228–241 (1999)
14. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: *Proc. 16th ACM Symp. SOSP 1997*, pp. 129–142 (1997)
15. Necula, G.C.: Proof-carrying code. In: *Proc. 24th ACM SIGPLAN-SIGACT Symp. POPL 1997*, pp. 106–119 (1997)
16. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21(1), 5–19 (2003)
17. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: *Proc. 18th IEEE CSFW 2005*, pp. 255–269 (2005)
18. Simonet, V.: Flow Caml in a nutshell. In: *Proc. APPSEM-II*, pp. 152–165 (2003)
19. Win, B.D., Piessens, F., Smans, J., Joosen, W.: Towards a unifying view on security contracts. In: *Proc. SESS 2005*, pp. 1–7 (2005)
20. Zdancewic, S.: Challenges for information-flow security. In: *The 1st Intl. Workshop on Programming Language Interference and Dependence (PLID 2004)* (2004)

New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough

Chong Hee Kim* and Jean-Jacques Quisquater

UCL Crypto Group, Université Catholique de Louvain, Belgium
Place du Levant, 3, Louvain-la-Neuve, 1348, Belgium
{chong-hee.kim, Jean-Jacques.Quisquater}@uclouvain.be

Abstract. In this paper we show a new differential fault analysis (DFA) on the AES-128 key scheduling process. We can obtain 96 bits of the key with 2 pairs of correct and faulty ciphertexts enabling an easy exhaustive key search of 2^{32} keys. Furthermore we can retrieve the entire 128 bits with 4 pairs. To the authors' best knowledge, it is the smallest number of pairs to find the entire AES-128 key with a fault attack on the key scheduling process. Up to now 7 pairs by Takahashi et al. were the best. By corrupting state, not the key schedule, Piret and Quisquater showed 2 pairs are enough to break AES-128 in 2003. The advantage of DFA on the key schedule is that it can defeat some fault-protected AES implementations where the round keys are not rescheduled prior to the check. We implemented our algorithm on a 3.2 GHz Pentium 4 PC. With 4 pairs of correct and faulty ciphertexts, we could find 128 bits less than 2.3 seconds.

Index terms: Fault attack, Differential Fault Analysis, AES, DFA, AES key schedule.

1 Introduction

Boneh et al. introduced the *fault attack* on the implementation of RSA-CRT (Chinese Remainder Theorem) with the errors induced by the fault injection in September 1996 [5]. After that, many papers have been published on this subject. In October 1996, Biham and Shamir published a fault attack on secret key cryptosystems entitled *Differential Fault Analysis* (DFA) [2]. On the 2nd October 2000, the AES became the successor of the DES and since then, it has been used more and more in many applications. Several authors mounted DFA on AES [3,7,9,11]. They assumed that the intermediate states were corrupted by the fault injection and tried to find out the key. Among them, the attack by Piret and Quisquater is the most efficient [11]. Their attack only needs two pairs of correct and faulty ciphertexts to retrieve 128 bits of AES-128.

A different form of DFA, targeting the AES key schedule, was introduced by Giraud in [8] and improved by Chen and Yen [6]. However, Giraud's attack does

* Supported by Walloon Region, Belgium / E.USER project.

not target only the AES key schedule. He used the faults in the intermediate state as well. Recently Peacham and Thomas improved DFA on AES key scheduling [10]. They assumed that random faults are injected during the execution of the AES key scheduling process and that the resulting faults propagate to all keys after the injection. They reduced the number of correct and faulty ciphertexts to 12 pairs to recover 128-bit key. Takahashi et al. generalized Peacham and Thomas's attack and reduced the required number of pairs more [13]. They succeeded in recovering 128-bit key with 7 pairs.

In this paper, we propose a new DFA on AES key scheduling process. Our attack takes advantage of faults occurring in the 9th round of the AES key scheduling process. Thus the fault model and the hypothesis on the fault location are exactly the same as in Peacham and Thomas' and Takahashi et al.'s. However the way we exploit faults is different from theirs. We retrieved the entire 128-bit key of AES-128 with 4 pairs of correct and faulty ciphertexts. Two pairs are enough to recover 96 bits of the key enabling an easy exhaustive key search of the remaining 2^{32} keys. We implemented our algorithm on a 3.2 GHz Pentium 4 PC. With 4 pairs of correct and faulty ciphertexts we found 128 bits in less than 2.3 seconds.

The rest of this paper is organized as follows. In Section 2, we briefly describe AES. The review on the previous DFA on AES Key schedule is presented in Section 3. Our analysis methodology is presented in Section 4. Section 5 compares our attack with previous attacks, with the conclusion given in Section 6.

2 AES

AES [1] can encrypt and decrypt 128 bits of block with 128, 192, or 256 bits of key. In this paper we will only deal with the 128-bit key variant, AES-128, as it is the most widely used. Our attack can be extended trivially to other variants. The intermediate computation result of AES-128, called *State* is usually represented by a 4×4 matrix, each cell of which is a byte as shown in Fig. 1. Where, $S_{j,k}^i$ denotes $(j+1)^{th}$ row and $(k+1)^{th}$ column byte of i^{th} State, $j, k \in \{0, \dots, 3\}$.

In the rest of the paper, we will use the following additional notations:

- $K_{j,k}^i$ denotes $(j+1)^{th}$ row and $(k+1)^{th}$ column byte of i^{th} AES round key, $i \in \{0, \dots, 10\}$ and $j, k \in \{0, \dots, 3\}$,
- S^0 denotes the State after 9th *AddRoundKey*,
- S^1 denotes the State after 10th *SubBytes*,
- S^2 denotes the State after 10th *ShiftRows*,
- S^3 denotes the State after 10th *AddRoundKey*,
- S_j^i denotes 32-bit $(j+1)^{th}$ row of S^i , $j \in \{0, \dots, 3\}$,
- $(C, C^*), (D, D^*)$ denote the correct and faulty ciphertext pairs.

AES-128 has 10 rounds. Each round function is composed of 4 transformations except the last round: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*.

$S_{0,0}^i$	$S_{0,1}^i$	$S_{0,2}^i$	$S_{0,3}^i$
$S_{1,0}^i$	$S_{1,1}^i$	$S_{1,2}^i$	$S_{1,3}^i$
$S_{2,0}^i$	$S_{2,1}^i$	$S_{2,2}^i$	$S_{2,3}^i$
$S_{3,0}^i$	$S_{3,1}^i$	$S_{3,2}^i$	$S_{3,3}^i$

Fig. 1. The State during AES encryption

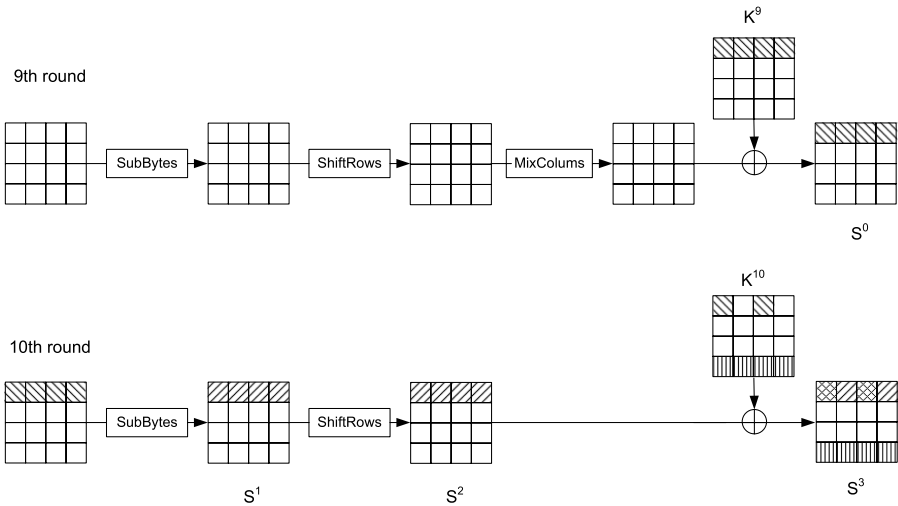


Fig. 2. Last two rounds of AES encryption

The last round is lacking *MixColumns*. Our attack focuses on the last two rounds. They are depicted in Fig. 2

SubBytes. It is made up of the application of 16 identical 8×8 S-boxes. This is a non-linear byte substitution. We denote the function of SubBytes **SB**. That is, $\mathbf{SB}(S^i) = \text{SubBytes}(S^i)$. For the simplicity, we define that **SB** also can take a byte and two bytes as an input as follows:

$$\begin{aligned} \mathbf{SB}(S_{j,k}^i) &= \text{Sbox}(S_{j,k}^i), \\ \mathbf{SB}(S_{j,k}^i, S_{j,l}^i) &= \text{Sbox}(S_{j,k}^i), \text{Sbox}(S_{j,l}^i). \end{aligned}$$

We denote *Inverse SubBytes* \mathbf{SB}^{-1} . We define that \mathbf{SB}^{-1} also can take bytes as an input.

ShiftRows. Each row of the *State* is cyclically shifted over different offsets. Row 0 is not shifted, row 1 is shifted by 1 byte, row 2 is shifted by 2 bytes, and row 3 by 3 bytes. We denote *ShiftRows* and its inverse, *InverseShiftRows*, \mathbf{SR} and \mathbf{SR}^{-1} respectively. We also define that they can take bytes as an input.

MixColumns. This is a linear transformation to each column of the *State*. Each column is considered as polynomial over \mathbb{F}_{2^8} and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x) = 03 * x^3 + 01 * x^2 + 01 * x + 02$.

AddRoundKey. It is a bitwise XOR with a round key.

We briefly describe the last two rounds, 9th and 10th rounds, of the key scheduling process as shown in Fig. 3. The input 128-bit key is divided into four 32-bit columns. The first 32-bit column propagates to the next column in the same round, which generates the second column. The second and third columns do the same. The fourth column propagates to the next round through the function RotWord, which performs a cyclic permutation and SubWord, which applies S-box. Each column generated in the key process is grouped to yield the 128-bit round key.

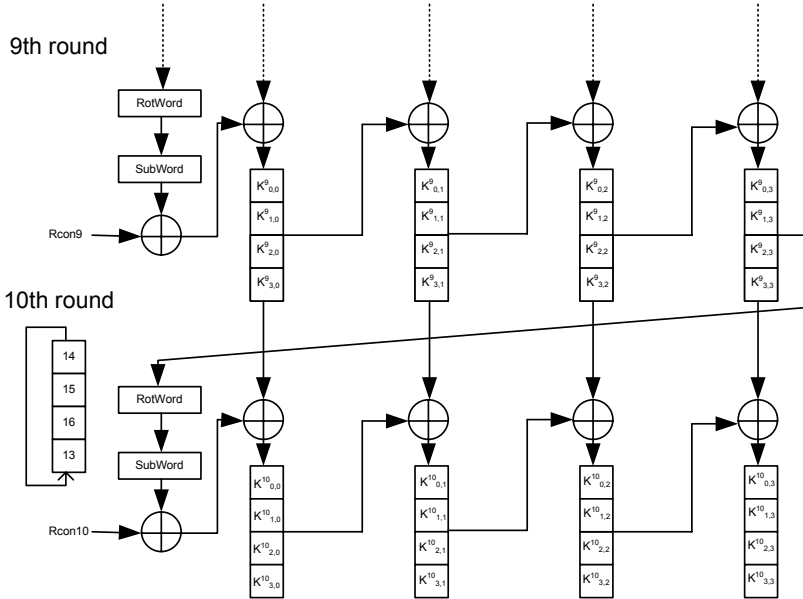


Fig. 3. 9th and 10th AES Key scheduling process

3 Previous Works about DFA on AES Key Schedule

The first DFA on AES key schedule was done by Giraud, but still needs to attack the intermediate state [8]. He presented two fault attacks on the AES. Both

require the ability to obtain several faulty ciphertexts originating from the *same* plaintext (contrary to our attack). The first one assumes it is possible to induce a fault on only one bit of an intermediate state. Under this condition, 50 faulty ciphertexts are necessary to retrieve the full key. The second attack exploits faults on bytes. It requires the ability of inducing faults at several chosen places both on key scheduling process and intermediate state. Therefore, the second is the first attempt of attack on key scheduling process, but it is not complete. Because it still needs to attack on the intermediate state. It could retrieve the key with 250 faulty ciphertexts. If he extends his hypothesis by supposing that the attacker can choose the byte affected by the fault, the first attack requires 35 faulty ciphertexts and the second requires 31 faulty ciphertexts.

In 2003, Chen and Yen improved Giraud’s attack [6]. They could retrieve the key with fewer faulty ciphertexts and with less computational complexity. Giraud’s second attack is composed of three steps, an attack on 9th round key, an attack on 8th round key, and an attack on 8th round intermediate state. The first two steps of Chen and Yen’s method are similar to Giraud’s. But the third step focuses on the *Inverse SubBytes* and requires less samples.

Unlike the previous two attacks, Peacham and Thomas assumes that random faults are injected during the execution of the AES key scheduling process and the resulting faults propagate to all keys after the injection [10]. They showed that 12 pairs of correct and faulty ciphertexts are enough to retrieve the whole key without brute-force search. They assumed that all bytes of a 32-bit column of the 9th round key are corrupted during the execution of the key scheduling process. Their attack consists of four steps. They use the fact that the intermediate state calculated by the correct ciphertext just before the *AddRoundKey* of the 9th round is equal to the intermediate state calculated by the faulty ciphertext just before the *AddRoundKey* of the 9th round.

In 2007, Takahashi et al. generalized Peacham and Thomas’s attack and reduced the required number of pairs [13]. They could retrieve the whole key with 7 pairs and 80 bits of the key with 2 pairs. Recently they improved their attack a little bit by assuming that faults are injected into 32 bits of one column [12]. They found 88 bits with 2 pairs but still they need 7 pairs to find 128 bits.

4 Our DFA on AES Key Schedule

In this section we describe our attack. After presenting our fault model, we describe our *basic attack* that retrieves 32 bits of the key with 2 pairs of correct and faulty ciphertexts giving a one byte random error on 9th round key scheduling process. Then we improve our attack by giving a random fault on three bytes of a 32-bit column of the 9th round key scheduling process. We can retrieve 96 bits of the key with 2 pairs, and all 128 bits with 4 pairs.

4.1 Fault Model

We assume that a random fault is induced in the 9th round of the AES key scheduling process and some bytes of the first column of the 9th round key are

corrupted. In addition, we assume that the attacker can obtain pairs of correct and faulty outputs from the same input. However we do not need the several faulty outputs with the same plaintext.

The fault model and hypothesis on the fault location are exactly the same as in Peacham and Thomas' and Takahashi et al.'s. However the way we exploit faults is different from them. We exploit the intermediate state *after* the *AddRoundKey* of the 9th round contrary to previous attacks. They exploit the intermediate state *before* the *AddRoundKey* of the 9th round. Secondly they try to find the 9th round key but we find the 10th round key. Finally we try to remove impossible candidates for 10th round key, but they try to find directly the correct 9th round key.

It is quite interesting to refer DFA on AES state. Piret and Quisquater's DFA on AES state [11] requires the minimum number of the ciphertexts even though they use the same fault model and the hypothesis on the fault location of Dusart et al.'s attack [7]. The way Dusart et al. exploit faults is quite similar to Peacham and Thomas' and Takahashi et al.'s. Dusart et al. write and solve a system of equations of which the unknown value is the one of the fault. Our way of exploiting faults follows the way of Piret and Quisquater's.

4.2 Basic Attack

We assume that one byte of the first column of the 9th round key is corrupted. For simplicity, we assume $K_{0,0}^9$ is corrupted into $\tilde{K}_{0,0}^9$. The attack can be applied when another byte is corrupted. We denote the difference between them as a , i.e., $a = K_{0,0}^9 \oplus \tilde{K}_{0,0}^9$. This error propagates to some bytes of the round keys as shown in Fig. 4. The four bytes of 9th round key, $(K_{0,0}^9, K_{0,1}^9, K_{0,2}^9, K_{0,3}^9)$, and six bytes of 10th round key, $(K_{0,0}^{10}, K_{0,2}^{10}, K_{3,0}^{10}, K_{3,1}^{10}, K_{3,2}^{10}, K_{3,3}^{10})$, are corrupted. This also results in the corruption on the intermediate states as shown in Fig. 2.

We exploit the intermediate state *after* the *AddRoundKey* of the 9th round, i.e., S^0 . We denote the i^{th} faulty state \tilde{S}^i . Only the first row of S^0 receives the effect of the faults. By doing XOR between S_0^0 and \tilde{S}_0^0 , we have (we remind that S_j^i is denoted as a 32-bit $(j+1)^{\text{th}}$ row of S^i):

$$\begin{aligned} S_0^0 \oplus \tilde{S}_0^0 &= \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_0 \oplus K_0^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_0^* \oplus \tilde{K}_0^{10})] \\ &= (a, a, a, a), \end{aligned} \quad (1)$$

where, C_0 is the 32-bit first row of the correct ciphertext, C_0^* is the 32-bit first row of the faulty ciphertext, K_0^{10} is the 32-bit first row of 10th correct round key, and \tilde{K}_0^{10} is the 32-bit first row of 10th faulty round key.

An error on a byte makes 255 possible differences. That is, $a \in \{1, 2, \dots, 255\}$. Therefore the number of the possible differences of S_0^0 and \tilde{S}_0^0 is 255. In equation (1), we know C_0 and C_0^* . Therefore we can eliminate the wrong candidates for K_0^{10} that do not meet the condition of equation (1).

We compute how many wrong candidates for the round key K_0^{10} can be removed by a single pair (C, C^*) with the equation (1). We define the difference of the first 32-bit row of C and C^* as Δ , i.e., $\Delta = C_0 \oplus C_0^*$. The number of

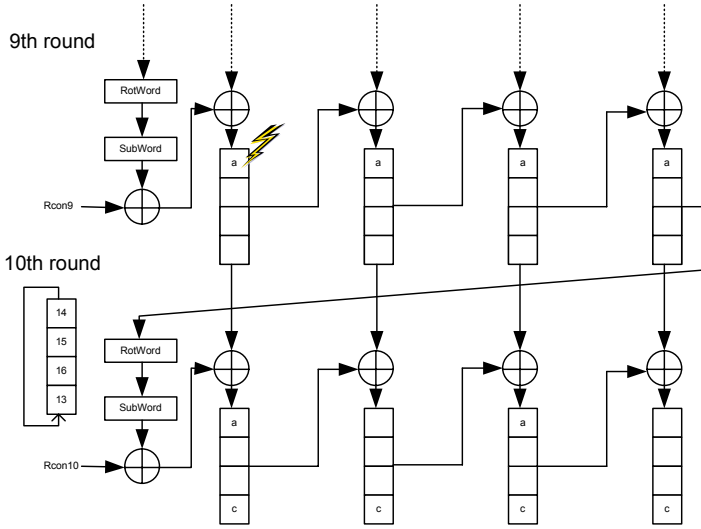


Fig. 4. 9th and 10th AES Key scheduling process with faults

possible value for Δ is 255^4 . Among them 255 differences satisfy the equation (II). Thus the fraction of the candidates for the round key K_0^{10} surviving the test with equation (II) is $255/255^4$. Therefore we conclude that the number of remaining wrong candidates for K_0^{10} after N pairs have been treated is about $256^4(255^{-3})^N$. With one pair, about 259 candidates remain. If two pairs are exploited, we are in principle left with the right candidate only.

In the above paragraph we assumed that we needed to guess K_0^{10} only and we knew other parameters. However, to solve the equation (II) we need to guess both K_0^{10} and \tilde{K}_0^{10} . Since we have 2^{64} candidates for $(K_0^{10}, \tilde{K}_0^{10})$, it is not practical to guess K_0^{10} and \tilde{K}_0^{10} together. However, K_0^{10} and \tilde{K}_0^{10} satisfy the following condition:

$$K_0^{10} \oplus \tilde{K}_0^{10} = (a, 0, a, 0).$$

That is, $K_{0,1}^{10} = \tilde{K}_{0,1}^{10}$ and $K_{0,3}^{10} = \tilde{K}_{0,3}^{10}$. Therefore we can start the attack with these two bytes that results in 2^{16} candidates instead of 2^{64} . The number of remaining candidates for $(K_{0,1}^{10}, K_{0,3}^{10})$ after N pairs of the correct and wrong ciphertexts is now $256^2(255^{-1})^N$. With two pairs, we are left with almost one candidate for $(K_{0,1}^{10}, K_{0,3}^{10})$.

This consideration leads to the following sketch of our *basic attack*. We need two pairs of the correct and faulty ciphertexts (C, C^*) and (D, D^*) . We do not need to have the same faulty value in $K_{0,0}^9$ for these two pairs. We define the

¹ Because $256^2(255^{-1})^2 = 1.0079$, we have more than one candidate left for $(K_{0,1}^{10}, K_{0,3}^{10})$ sometimes. However, after Step 2 we have only one candidate left since $256^3(255^{-2})^2 = 0.004$ for $(K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10})$.

error during the computation of C^* as a_1 , i.e., $a_1 = K_{0,0}^9 \oplus \tilde{K}_{0,0}^9$ and the error during the computation of D^* as a_2 , i.e., $a_2 = K_{0,0}^9 \oplus \tilde{K}_{0,0}^9$ for (D, D^*) .

We first find the candidates for $(K_{0,1}^{10}, K_{0,3}^{10})$ with two pairs of the correct and faulty ciphertexts. Normally after this step, we have 1 or 2 candidates. Then we find the candidates for $(K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10})$. Finally we find the candidates for $(K_{0,0}^{10}, K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10})$.

Step 1. We compute the candidate for $(K_{0,1}^{10}, K_{0,3}^{10}, a_1, a_2)$. The inputs for this step are two pairs of the correct and faulty ciphertexts (C, C^*) and (D, D^*) .

Algorithm 1

1. Set up a list \mathcal{L} containing all 2^{16} candidates for $(K_{0,1}^{10}, K_{0,3}^{10})$.
2. Choose a candidate from \mathcal{L} and compute (α_1, α_2) and (β_1, β_2) as follows:
 $(\alpha_1, \alpha_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,1} \oplus K_{0,1}^{10}, C_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,1}^* \oplus K_{0,1}^{10}, C_{0,3}^* \oplus K_{0,3}^{10})]$ and $(\beta_1, \beta_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,1} \oplus K_{0,1}^{10}, D_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,1}^* \oplus K_{0,1}^{10}, D_{0,3}^* \oplus K_{0,3}^{10})]$.
3. Add the candidate and (α_1, β_1) to a new list \mathcal{M} if $\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2$.
4. Repeat Step 2 and Step 3 for all candidates from \mathcal{L} .

Finally \mathcal{M} has the candidates for $(K_{0,1}^{10}, K_{0,3}^{10}, a_1, a_2)$.

Step 2. We compute the candidate for $(K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10}, a_1, a_2)$. The inputs for this step are (C, C^*) , (D, D^*) , and the list \mathcal{M} from Step 1. We note that $\tilde{K}_{0,2}^{10}$ can be computed as $\tilde{K}_{0,2}^{10} = K_{0,2}^{10} \oplus a$.

Algorithm 2

1. Set up a list \mathcal{L} containing all 2^8 candidates for $K_{0,2}^{10}$.
2. Choose a candidate from \mathcal{L} .
3. Choose a candidate from \mathcal{M} and compute (α_1, α_2) and (β_1, β_2) as follows:
 $(\alpha_1, \alpha_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,2} \oplus K_{0,2}^{10}, C_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,2}^* \oplus K_{0,2}^{10} \oplus a_1, C_{0,3}^* \oplus K_{0,3}^{10})]$ and $(\beta_1, \beta_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,2} \oplus K_{0,2}^{10}, D_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,2}^* \oplus K_{0,2}^{10} \oplus a_2, D_{0,3}^* \oplus K_{0,3}^{10})]$.
4. Add $(K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10}, a_1, a_2)$ to a new list \mathcal{N} if $\alpha_1 = \alpha_2 = a_1$ and $\beta_1 = \beta_2 = a_2$.
5. Repeat Step 3 and Step 4 for all candidates from \mathcal{M} .
6. Repeat from Step 2 to Step 5 for all candidates from \mathcal{L} .

Finally \mathcal{N} has the candidates for $(K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10}, a_1, a_2)$.

Step 3. We compute the candidate for $(K_{0,0}^{10}, K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10}, a_1, a_2)$. The inputs for this step are (C, C^*) , (D, D^*) , and the list \mathcal{N} from Step 2. We note that $\tilde{K}_{0,0}^{10}$ can be computed as $\tilde{K}_{0,0}^{10} = K_{0,0}^{10} \oplus a$.

Algorithm 3

1. Set up a list \mathcal{L} containing all 2^8 candidates for $K_{0,0}^{10}$.
2. Choose a candidate from \mathcal{L} .

3. Choose a candidate from \mathcal{N} and compute (α_1, α_2) and (β_1, β_2) as follows:
 $(\alpha_1, \alpha_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,0} \oplus K_{0,0}^{10}, C_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_{0,0}^* \oplus K_{0,0}^{10} \oplus a_1, C_{0,3}^* \oplus K_{0,3}^{10})]$ and $(\beta_1, \beta_2) = \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,0} \oplus K_{0,0}^{10}, D_{0,3} \oplus K_{0,3}^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(D_{0,0}^* \oplus K_{0,0}^{10} \oplus a_2, D_{0,3}^* \oplus K_{0,3}^{10})]$.
4. Output $(K_{0,0}^{10}, K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10}, a_1, a_2)$ and stop the algorithm if $\alpha_1 = \alpha_2 = a_1$ and $\beta_1 = \beta_2 = a_2$.
5. Repeat Step 3 and Step 4 for all candidates from \mathcal{N} .
6. Repeat from Step 2 to Step 5 for all candidates from \mathcal{L} .

Finally we have the one correct key for $(K_{0,0}^{10}, K_{0,1}^{10}, K_{0,2}^{10}, K_{0,3}^{10})$ and faulty values (a_1, a_2) .

We implemented our attack on a 3.2 GHz Pentium 4 PC and found K_0^{10} with about 0.5 second. If we give faults in $K_{1,0}^9$ instead of $K_{0,0}^9$, then similarly we can find K_1^{10} . Therefore with 8 pairs, we can find the entire 128 bits of 10^{th} round key. We can easily compute the initial key with 10^{th} round key, see [7].

4.3 Improved Attack

Now let us consider the errors on more than one byte on the first column of 9^{th} round key. We suppose that the first two bytes, $K_{0,0}^9$ and $K_{1,0}^9$, are corrupted by fault injection. Let us denote that $a = K_{0,0}^9 \oplus \tilde{K}_{0,0}^9$ and $b = K_{1,0}^9 \oplus \tilde{K}_{1,0}^9$. According to AES key scheduling process, these differences a and b make another difference c and d in 10^{th} round key respectively as shown in Fig. 5.

Two-byte error makes two rows of S^0 to be corrupted. We define the difference in the second row, S_1^0 , as (b, b, b, b) . The corresponding difference in the second row of the 10^{th} round key, K_1^{10} , is $(b, 0, b, 0)$. Therefore we can find K_1^{10} and b with the *basic attack* in Sec. 4.2 with the following condition:

$$\mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_1 \oplus K_1^{10})] \oplus \mathbf{SB}^{-1}[\mathbf{SR}^{-1}(C_1^* \oplus \tilde{K}_1^{10})] = (b, b, b, b). \quad (2)$$

Then we can compute d from the value of b and K_1^{10} (this comes from the structure of the AES keys scheduling process) as follows:

$$\begin{aligned} K_{1,3}^9 &= K_{1,2}^{10} \oplus K_{1,3}^{10}, \\ d &= \text{SBox}(K_{1,3}^9) \oplus \text{SBox}(K_{1,3}^{10} \oplus b). \end{aligned}$$

Because we know the value of d , we only do not know the value of a in the first row of difference of K^{10} as shown in (c) of Fig. 5. Therefore we can apply the *basic attack* to the first row and find K_0^{10} .

We summarize how to find 64 bits of K^{10} as follows:

Algorithm 4

1. Compute K_1^{10} and (b_1, b_2) using *basic attack*.
2. Compute d_1, d_2 with $(K_{1,2}^{10}, K_{1,3}^{10})$ and b_1, b_2 .
3. Compute K_0^{10} and (a_1, a_2) using *basic attack*.

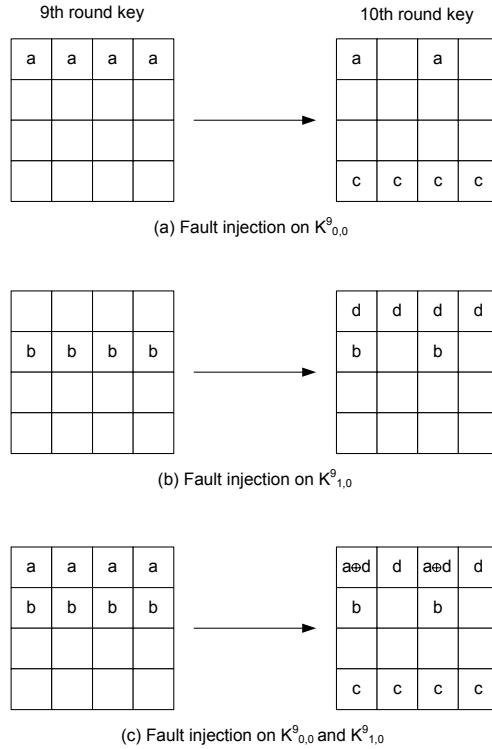


Fig. 5. Differences between correct and wrong round keys according to fault injection on the 9^{th} round key

We can further improve the attack in case three bytes of the first column of 9^{th} round key are corrupted. As shown in Fig. 6, let us denote $e = K_{2,0}^9 \oplus \tilde{K}_{2,0}^9$. We first start with the third row. With the basic attack, we compute K_2^{10} and e . Then we compute f with the property of AES key scheduling process. We can compute K_1^{10} and K_0^{10} with Algorithm 4. Therefore we can compute 96 bits of AES-128 key with two pairs of correct and faulty ciphertexts. We can compute

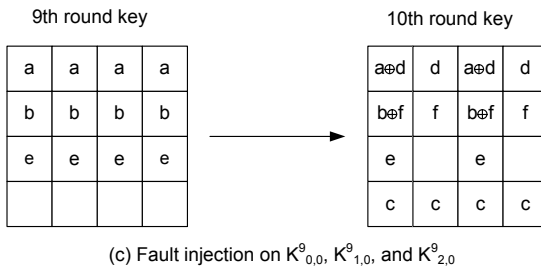


Fig. 6. Differences between correct and wrong round keys in case of the fault injection on the three bytes of the 9^{th} round key

the last 32 bits with an exhaustive search or with the basic attack of one byte fault on $K_{3,0}^9$ and another two pairs of correct and faulty ciphertexts.

We again implemented our improved attack on the same PC. To find 96 bits of the key with 2 pairs, average 1.8 seconds are required. To compute 128 bits with 4 pairs, it requires 2.3 seconds in average.

5 Comparison with Previous Attacks

We compared our attack with previous attacks in terms of the relation between the retrieved bits of key and the required number of pairs as shown in Fig. 7.

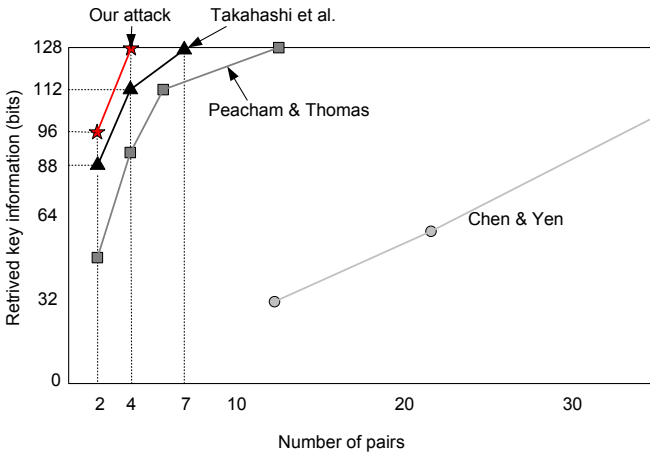


Fig. 7. Comparison in terms of required number of pairs

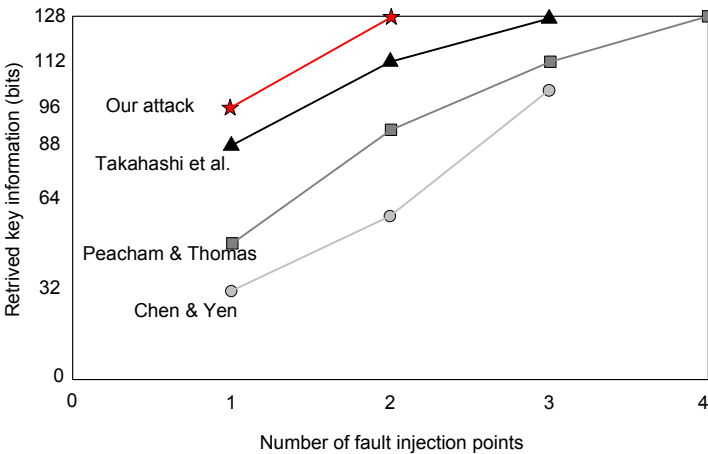


Fig. 8. Comparison in terms of required number of fault injection points

We also compared in terms of the number of fault injection points in Fig. 8. In both cases, we can see our proposed method is the best.

If we have only two pairs of the correct and wrong ciphertexts, we can compute 96 bits with our attack and need to do an exhaustive search for the other 32 bits. We estimated the time for the exhaustive search based on the simulation result of [13]. On a normal PC, the estimated calculation time of the 32-bit exhaustive search is about 8 minutes. If we use Takahashi et al.'s attack, we need to do an 40-bit exhaustive search, which requires about 3 days.

6 Conclusions

We proposed a new differential fault analysis on AES key schedule. Only two pairs of correct and faulty ciphertexts are enough to find the whole key of AES-128 with DFA on AES state by Piret and Quisquater. In the area of DFA on AES key schedule, still we needed many pairs. However our proposed method reduced the gap between DFA on state and DFA on key schedule. Ours requires two pairs for retrieving 96 bits of the key enabling an easy exhaustive key search of 2^{32} keys and four pairs for 128 bits without an exhaustive key search. Our result shows the minimum number of pairs and that of fault injection points than the previous attacks. It takes about 2 seconds to retrieve 128 bits with four pairs on the normal PC. With two faults it takes about 8 minutes to find 128 bits.

The general countermeasure against DPA on AES state is to recompute the last three rounds of AES and compare it with the original output. However, if the key schedule is not re-done for the re-computation of the last three rounds it cannot prevent DPA on AES key schedule. Therefore we can conclude that key scheduling process as well as encryption process should be protected against fault attacks.

References

1. National institute of standards and technology, Advanced Encryption Standards. NIST FIPS PUB 197 (2001)
2. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
3. Blömer, J., Seifert, J.-P.: Fault based cryptanalysis of the advanced encryption standard (AES). In: Wright, R.N. (ed.) FC 2003. LNCS, vol. 2742, pp. 162–181. Springer, Heidelberg (2003)
4. Boneh, D., DeMillo, R., Lipton, R.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
5. Boneh, D., DeMillo, R., Lipton, R.: On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology* 14(2), 101–119 (2001); An earlier version appears in [4]

6. Chen, C.-N., Yen, S.-M.: Differential fault analysis on AES key schedule and some countermeasures. In: Safavi-Naini, R., Seberry, J. (eds.) ACISP 2003. LNCS, vol. 2727, Springer, Heidelberg (2003)
7. Dusart, P., Letourneux, G., Vivolo, O.: Differential fault analysis on A.E.S (2003)/10, <http://eprint.iacr.org/>
8. Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2004. LNCS, vol. 3373, pp. 27–41. Springer, Heidelberg (2005)
9. Moradi, A., Shalmani, M.T.M., Salmasizadeh, M.: A generalized method of differential fault attack against AES cryptosystem. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 91–100. Springer, Heidelberg (2006)
10. Peacham, D., Thomas, B.: A DFA attack against the AES key schedule. SiVenture White Paper 001 (26 October 2006), http://www.siventure.com/pdfs/AES_KeySchedule_DFA_whitepaper.pdf
11. Piret, G., Quisquater, J.-J.: A differential fault attack technique against SPN structures, with application to AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
12. Takahashi, J., Fukunaga, T.: Differential fault analysis on the AES key schedule. IACR Eprint archive 2007-480
13. Takahashi, J., Fukunaga, T., Yamakoshi, K.: DFA mechanism on the AES key schedule. In: Proc. of the Fourth International Workshop, FDTC 2007, pp. 62–72 (2007)

DSA Signature Scheme Immune to the Fault Cryptanalysis

Maciej Nikodem

The Institute of Computer Engineering, Control and Robotics
Wrocław University of Technology
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland
maciej.nikodem@pwr.wroc.pl

Abstract. In this paper we analyse the Digital Signature Algorithm (DSA) and its immunity to the fault cryptanalysis that takes advantage of errors inducted into the private key a . The focus of our attention is on the DSA scheme as it is a widely adopted by the research community, it is known to be vulnerable to this type of attack, but neither sound nor effective modifications to improve its immunity have been proposed. In our paper we consider a new way of implementing the DSA that enhances its immunity in the presence of faults. Our proposal ensures that inducting errors into the private key has no benefits since the attacker cannot deduce any information about the private key given erroneous signatures. The overhead of our proposal is similar to the overhead of obvious countermeasure based on signature verification. However, our modification generates fewer security issues.

1 Introduction

In recent years a variety of implementations based on tamper-proof devices (e.g. smart cards) have been proposed in order to provide better support for data protection. The main reason for this trend originate from the fact that such devices are expected to be characterized by high reliability and security. This is obtained thanks to their ability to perform complex arithmetical operations, to control incoming and outgoing communication, and to prevent unauthorized access. Cryptographic devices, on the other hand, are endangered by faults which can compromise their security.

In 1997 Bao et al. [3] and Boneh et al. [6] showed that if faults occur when the device performs cryptographic operation, then they may decrease security and leak the key stored inside of the device. The described problem has been presented for most of modern cryptographic algorithms such as the RSA encryption and signature scheme, identification protocols, and signature schemes based on the discreet logarithm problem (e.g. ElGamal, Schnorr, and DSA). The same year, Biham and Shamir [4] demonstrated that secret key cryptosystems are vulnerable to the fault cryptanalysis as well.

Since 1997, many researchers have been investigating the problem of fault cryptanalysis, in an effort to discover methods of enhancing security of different

cryptographic schemes. However, a handful solutions have been proposed only for the RSA algorithm [2][2][21] and symmetric key cryptosystems [9][10][13]. Less attention has been paid to signature schemes, of which security is based on the Discrete Logarithm Problem (DLP).

All of existing solutions can be divided into three groups:

- fault prevention,
- error detection and error correction,
- error diffusion, also referred to as *ineffective computation* [21].

The main purpose of using fault prevention techniques is to minimise the probability of fault injection. This is achieved by hardware circuits responsible for shielding a device, and protecting it from reverse engineering, radiation, ion beams, power spikes, and signal glitches [1][16]. Unfortunately, none of these methods work properly – some have design bugs while others can be simply switched off [16]. If fault prevention fails then the device is susceptible to errors, and other types of countermeasures have to be taken.

Error detection and error correction aim to detect and report an error in order to prevent successful cryptanalysis. Existing error detection schemes are based on well-known error detection techniques that utilise reverse computations (e.g. ciphertext decoding or signature verification) or parity checks and residue codes. Such simple solutions allow to enhance immunity to the fault analysis, and may be easily implemented in symmetric encryption algorithms [14]. Some steps to protect the RSA algorithm have been taken too [8]. Unfortunately reverse computations usually introduce delays and computation overhead which may be impractical. Moreover both methods require comparing procedure, that actually verifies whether computations were error free or not, which is may be a target of fault injection, and therefore, may become a bottleneck of the whole solution [15][20][21]. This threat is not present when the error correction is used, since it is performed regardless of if errors have been inducted or not, and thus requires no comparisons. Unfortunately, only one error correction scheme for the AES algorithm has been proposed so far [9].

The error diffusion is another method of preventing fault cryptanalysis. In contrast to previous solutions it attempts to modify a cryptographic algorithm in such a way that any inducted error is spread among different cryptographic computations and the output. The goal of the error diffusion is to produce erroneous output that is useless for the attacker. This is possible, since most fault attack scenarios assume that the attacker induces particular types of errors. Moreover, it is often required that errors are inserted into selected part of the algorithm. Changing and dispersing the error increases the overhead of fault cryptanalysis causing such attacks to be ineffective. For that reason this type of countermeasure is also referred to as ineffective computations. Although error diffusion was proposed five years ago [21], so far it has been adopted only for the RSA-CRT algorithm. In last year a number of different error diffusion techniques for the RSA-CRT were proposed (e.g. [5][8][21]) but most of them have been broken [15][22]. Moreover, no countermeasures of this type have been proposed for other cryptographic algorithms.

2 Related Work

A feasibility for implementation of the fault cryptanalysis in case of DLP-based signature schemes was first announced in 1997 [3]. Attack described in this paper utilises bit-flip errors inducted into random bit of the private key a . Due to such error the resulting signature is affected with error that may be effectively guessed by the attacker at the cost of $2n$ modular exponentiations. This result was put forth in 2000 by Dottax who presented how one can implement fault cryptanalysis to ECDSA. In 2004, Giraud and Knudsen [11] extended previous results and analysed an attack on the DSA scheme that take advantage of byte errors instead of bit errors. Paper [11] presents detailed analysis of attack complexity, and the number of faulty signatures required to restrict possible private key values to the requested amount. On the other hand, it does not describe any countermeasures that can be applied in order to improve security.

The security of the DSA scheme was also analysed by Rosa [19] who presented a lattice-based fault attack. This attack can be carried out irrespectively of the fact whether a tamper-proof device performs result checking before output or not. It requires for the attacker to substitute g used during signing procedure with $g' = g\beta \bmod p$ for some $1 < \beta < p$ with $\text{ord}(\beta) = d$ in Z_p^* and $\text{gcd}(d, q) = 1$. This can be difficult to achieve with fault induction, when the DSA is implemented in hardware but, as presented in [19], it can be performed if the scheme is implemented in software. An obvious countermeasure for software implementation, as presented in [19], is to make manipulation on the parameters of the DSA scheme impossible.

The lattice-based fault attack was also presented by Naccache et al. [17]. Their attack is based on faults inducted into random integer k in order to force a number of the least significant bytes (LSBs) of k to flip to 0. Afterwards the attacker applies lattice attack on the ElGamal-like signature which can recover private key, given sufficiently many signatures such that a few bits of corresponding k are zeroed. As presented in [17], when one LSB of each k is zeroed, then 27 signatures are sufficient to disclose the private key. In their paper Naccache et al. presented theory and methodology of the attack as well as possible countermeasures (e.g. checksums, randomisation or refreshments).

Although DLP-based signature algorithms are known to be prone to fault cryptanalysis, neither sound nor effective countermeasures have been proposed for these so far. Precisely, there exist only one obvious solution that utilises signature verification procedure in order to check whether generated signature is correct or not. If verification fails then either signing or verification was affected with faults and the signature has to be rejected. This countermeasure seems to work but in fact it utilises comparison procedure which is susceptible to fault attacks and thus may become a bottleneck of the whole proposal [15,20].

In this paper we give a practical method of how to increase immunity of the DSA scheme in the presence of faults that affect private key a . We assume that the pseudo random generator that chooses k operates correctly while the

attacker is able to induce random bit-flip or byte change errors into a . Our proposal is based on error diffusion which ensures that any error e , inducted into a , is spread within the signature s . This results in erroneous signature \bar{s} that is affected with error E , which depends both on e and unknown random integer k . Relation between E , e and k ensures that E can neither be computed nor effectively guessed by the attacker. Therefore, in order to perform the attack, one has to verify all of 2^{160} possible values of k one by one which cause the whole attack to be ineffective. Immunity to the fault cryptanalysis is attained at the expense of the increased computational overhead which is similar to the overhead of signature verification. Relatively small overhead enables to implement the proposed scheme in small cryptographic devices like smart cards.

3 DSA Signature Scheme

The DSA signature scheme was proposed in 1991 by U.S. National Institute of Standards and Technology (NIST), and became first digital signature standard (DSS) ever recognised by any government. The DSA is a variant of the ElGamal signature scheme which requires a hash function $h : \{0, 1\}^* \rightarrow Z_q$ for prime integer q . Its security is based on the discreet logarithm problem.

Key generation in the DSA scheme is done as follows:

- select a prime number q such that q is 160 bit long ($2^{159} < q < 2^{160}$),
- choose t so that $0 \leq t \leq 8$, and select a prime number p where $2^{511+64t} < p < 2^{512+64t}$ with the property that $q|(p-1)$,
- select a generator g of the unique cyclic group of order q in Z_p^* ,
- select a random integer a such that $1 \leq a \leq q-1$,
- compute $y = g^a \bmod p$,
the public key is $\langle p, q, g, y \rangle$, and the private key is a .

After key generation the device stores the private key a and system parameters p, q, g for future use. Signing for a given message m goes as follows:

- select a random integer $0 < k < q$,
- compute $r = (g^k \bmod p) \bmod q$,
- compute $s = k^{-1} (h(m) + ar) \bmod q$,
the signature for m is a pair $\langle r, s \rangle$.

Given the message m , signature $\langle r, s \rangle$, and the public key $\langle p, q, g, y \rangle$ one can verify whether the signature is actually correct. This is done as follows:

- compute $w = s^{-1} \bmod q$,
- compute $v_1 = g^{h(m)w} \bmod p$ and $v_2 = y^{rw} \bmod p$,
- verify if $v_1 \cdot v_2 \bmod p \bmod q \stackrel{?}{=} r$.

If last equation holds, then the signature is accepted, otherwise the signature is rejected.

4 Fault Cryptanalysis of the DSA Signature Scheme

As presented in [3,11,17,19] hardware implementations of the DSA scheme can be compromised with fault cryptanalysis. This can be done in a few ways: by affecting random integer k [17], public parameter g [19] or by inducting errors into the private key register during signing procedure [3,11]. In the remaining part of the paper we focus on the fault cryptanalysis that attempts to affect a private key a . Its purpose is to generate a faulty signature which is then used to deduce the key. Let us briefly present an attack scenario described in [3].

Assume that the attacker has a possibility to induct random bit-flip errors into the private key register. Since errors are inducted randomly and the private key register is of the size $n = \log_2 a$ so the probability that i -th bit will be affected with error equals $1/n$. Moreover, inducting exactly one bit-flip error into the i -th bit of the register cause a change of the private key which is now equal to $\bar{a} = a \pm 2^i$ where sign \pm depends on the original value of this bit.

The erroneous signing procedure executes as follows:

- select a random integer $0 < k < q$,
 - compute $r = (g^k \bmod p) \bmod q$,
 - compute $\bar{s} = k^{-1} (h(m) + \bar{a}r) \bmod q$,
- erroneous signature for m is a pair $\langle r, \bar{s} \rangle$.

Because the error has changed the private key into $\bar{a} = a \pm 2^i$, therefore, an element \bar{s} of the erroneous signature $\langle r, \bar{s} \rangle$ is equal

$$\begin{aligned} \bar{s} &= k^{-1} (h(m) + \bar{a}r) \bmod q = k^{-1} (h(m) + (a \pm 2^i) r) \bmod q \\ &= k^{-1} (h(m) + ar) \pm 2^i r k^{-1} \bmod q \\ &= s \pm 2^i r k^{-1} \bmod q. \end{aligned} \quad (1)$$

Due to (1) and because the fact attacker knows $r = g^k \bmod p \bmod q$ the fault analysis can be performed. This is done in a similar way as for the signature verification:

- compute $w = \bar{s}^{-1} \bmod q$,
- compute $v_1 = g^{h(m)w} \bmod p$ and $v_2 = y^{rw} (g^{rw})^{\pm 2^i} \bmod p$,
- look for the fault value $\pm 2^i$ for which following equation holds

$$r = v_1 v_2 \bmod p \bmod q = g^{h(m)w} y^{rw} (g^{rw})^{\pm 2^i} \bmod p \bmod q. \quad (2)$$

According to (2), in order to perform cryptanalysis the attacker needs to find a value of the inducted fault $\pm 2^i$. Since we assume errors are inducted at random hence the attacker does not know which bit of the private key was flipped. Therefore, to perform cryptanalysis the attacker needs to check all possible fault values and find the one for which (2) holds. The time required to perform this attack is dominated by $2n$ exponentiations that have to be computed.

Each iteration of the above procedure allows the attacker to recover one bit of the private key. Attacker can then repeat this procedure to get remaining bits,

but since errors are inducted at random it could happen that successive errors affect bits already known. This is a difficulty that increases attack overhead but as presented in [6] the attacker can repeat above procedure as long as demanded amount of the private key bits is known. Afterwards he can perform exhausting search in order to find remaining bits.

5 DSA Scheme Immune to the Fault Cryptanalysis

As mentioned in the previous section, the fault analysis countermeasure proposed in this paper is based on the error diffusion. The purpose of this solution is to check whether the public key a , used during signature generation, was affected with error or not. This has to be done without any comparisons and conditional operations since these may be a bottleneck of the whole proposal, similarly as in case of error detection schemes. Therefore, we propose to use public key y in order to verify the correctness of a . The outcome of this verification is called an error diffusion term T , and it is equal zero only if all computations were error free. Later on this term is used in signature generation in such a way that for $T = 0$ the signature s is correct while for $T \neq 0$ erroneous value of \bar{s} depends both on error inducted e and random integer k . The relation between \bar{s} , e , and k ensures that the attacker will find it difficult to guess the error e given erroneous signature $\langle r, \bar{s} \rangle$ for message m .

Implementation of the above countermeasure requires that computation of the signature s is split into two steps

$$v = k + ar \bmod q, \quad (3)$$

$$s = k' (h(m) + v) - 1 \bmod q. \quad (4)$$

These two steps are separated by one additional and one modified computation:

- additional computation of error diffusion term T

$$T = (y^{-r} g^v \bmod p \bmod q) - r \bmod q, \quad (5)$$

- modified computation of multiplicative inverse

$$k' = (k \oplus T)^{-1} \bmod q. \quad (6)$$

With these modifications the whole signing procedure goes according to the scheme [1].

According to the 4-th step of the proposed signature scheme the error diffusion term $T = 0$ if and only if

$$y^{-r} g^v \bmod p \bmod q = r. \quad (7)$$

If the attacker inducts an error e into the private key a , then the erroneous value of \bar{v} equals

$$\bar{v} = k + \bar{a}r \bmod q = k + (a + e)r \bmod q = k + ar + er \bmod q. \quad (8)$$

Scheme 1. Proposed modification of the DSA scheme

Require: message m , private key, and public key of the DSA scheme

Ensure: the DSA signature $\langle r, s \rangle$

- 1: select a random integer k with $0 < k < q$,
 - 2: compute $r = g^k \bmod p \bmod q$,
 - 3: compute $v = k + ar \bmod q$,
 - 4: compute $T = (y^{-r} g^v \bmod p \bmod q) - r \bmod q$,
 - 5: compute $k' = (k \oplus T)^{-1} \bmod q$,
 - 6: compute $s = k'(h(m) + v) - 1 \bmod q$
-

Due to faulty value of \bar{v} left side of (7) equals

$$\begin{aligned} y^{-r} g^{\bar{v}} \bmod p \bmod q &= g^{-ar} g^{v+er} \bmod p \bmod q = g^{-ar} g^{k+ar+er} \bmod p \bmod q \\ &= g^{k+er} \bmod p \bmod q. \end{aligned} \tag{9}$$

This equals r if and only if

$$k + er \bmod q = k, \tag{10}$$

which is equivalent to

$$er \bmod q = 0. \tag{11}$$

However, $r < q$ since it was computed modulo q , and e is not a multiplicity of q – otherwise (i.e. $q|e$) $\bar{a} = a + e \bmod q = a$ and no error affects the private key. Therefore, $er \bmod q \neq 0$ for any error e , and hence, $T = 0$ only if no errors were inducted, and $T \neq 0$ otherwise. It is worth to mention that non-zero value of T equals

$$T = g^{k+er} \bmod p \bmod q - r \bmod q = g^{er} \bmod p \bmod q. \tag{12}$$

According to (12) and assuming that the attacker inducts particular type of errors (e.g. bit-flip errors) he is able to guess possible values of $T = g^{er} \bmod p \bmod q$, given erroneous signature $\langle r, \bar{s} \rangle$. This information, however, do not simplify the attack considerably.

Error diffusion in the proposed scheme is obtained by the modified inverse computation

$$k' = (k \oplus T)^{-1} \bmod q. \tag{13}$$

Because inverse is a nonlinear transformation thus the value of k' depends on the term T and the random integer k in a nonlinear way. Therefore, if $T \neq 0$ then $\bar{k}' = k^{-1} + E$ where E is a non-linear function of T and k . Since k is unknown to the attacker, he cannot compute the error E even if he knows the value of T . Moreover, there are $q-2$ possible values of E because k is chosen randomly every iteration of the scheme. Finally, using \bar{k}' for the computation of the signature s yields its erroneous value \bar{s} to be affected both with error e and E . Therefore, to perform a cryptanalysis the attacker needs to guess both errors, and since there are 2^{160} possible values of E such attack is infeasible.

On the other hand, if no errors were inducted into the private key a , then the proposed scheme outputs correct DSA signature. This is quite obvious since in such situation we obtain

$$\begin{aligned} T &= (y^{-r} g^v \bmod p \bmod q) - r \bmod q = (g^{-ar} g^{k+ar}) \bmod p \bmod q - r \bmod q \\ &= g^k \bmod p \bmod q - r \bmod q = 0, \end{aligned} \quad (14)$$

and

$$k' = (k \oplus T)^{-1} \bmod q = k^{-1} \bmod q. \quad (15)$$

This gives the signature equal

$$\begin{aligned} s &= k' (h(m) + v) - 1 \bmod q = k^{-1} (h(m) + k + ar) - 1 \bmod q \\ &= k^{-1} (h(m) + ar) \bmod q, \end{aligned} \quad (16)$$

which is a standard DSA signature.

6 Security of the Proposed Scheme

The attack scenarios presented in [3,11] assume that the attacker inducts particular types of errors. This allows him to guess inducted error effectively, and use this knowledge to restrict the number of possible private keys.

In our proposal erroneous signature \bar{s} is affected with error e and E , where the value of E depends on inducted error e and random integer k , chosen by the device. Because k is unknown and cannot be computed by the attacker, thus he can perform no better than guessing. Precisely, for each possible value of e the attacker has to find t for which following equation holds

$$\left(g^{h(m)} y^r g^{er} \right)^{\bar{s}^{-1}} g^t \bmod p \bmod q = r. \quad (17)$$

Taking into account that $\bar{s} = k' (h(m) + ar + er) \bmod q$, $k' = (k \oplus T)^{-1} = k^{-1} + E$ and assuming that the attacker guessed the inducted error e correctly, we can simplify the above equation

$$g^{(k^{-1}+E)^{-1}} g^t \bmod p \bmod q = g^{k \oplus T + t} \bmod p \bmod q = r. \quad (18)$$

Equation (18) shows that the attacker has to guess proper value of t such that

$$k \oplus T + t \bmod q = k, \quad (19)$$

or equivalently

$$k \oplus (g^{er} \bmod p \bmod q) + t \bmod q = k. \quad (20)$$

Accordingly, the sought value of t is a function of e and k . Furthermore, because k is chosen at random from the set $(1, q)$, there are $q - 2$ possible values of t for each e . Therefore, to perform an attack and to recover partial information on the private key a , the attacker has to guess the inducted error e and find t

that satisfies (18). This requires at least $n2^{161}$ exponentiations, assuming that the attacker induces single bit-flip errors.

A careful reader may realise that the proposed modification may be simplified since some surplus operations are performed. In fact, the proposed modification can be simplified, and will still work if we substitute equations (3-5) with following

$$v = ar \text{ mod } q, \tag{21}$$

$$s = k' (h(m) + v) \text{ mod } q, \tag{22}$$

$$T = y^{-r} g^v \text{ mod } p \text{ mod } q, \tag{23}$$

which are created by simply removing k , 1 and r from equations (3), (4), and (5) respectively. According to these changes the simplified signing goes according to the scheme 2.

Scheme 2. Simplified modification of the DSA scheme

Require: message m , private key, and public key of the DSA scheme

Ensure: the DSA signature $\langle r, s \rangle$

- 1: select a random integer k with $0 < k < q$,
 - 2: compute $r = g^k \text{ mod } p \text{ mod } q$,
 - 3: compute $v = ar \text{ mod } q$,
 - 4: compute $T = y^{-r} g^v \text{ mod } p \text{ mod } q$,
 - 5: compute $k' = (k \oplus T)^{-1} \text{ mod } q$,
 - 6: compute $s = k' (h(m) + v) \text{ mod } q$
-

Let us now briefly analyse the security of the simplified scheme. It is a quite simple task to verify that this simplified version has properties similar to the previous scheme (scheme 1):

- for any error e inducted into the private key a the error diffusion term $T = y^{-r} g^{ar+er} \text{ mod } p \text{ mod } q = g^{er} \text{ mod } p \text{ mod } q = 0$ only if $er \text{ mod } q = 0$. Similarly to the scheme 1 this holds only if $e = 0$,
- for any error e , k' is affected with error E that depends on e and the random integer k . Since E may not be computed thus the attacker can do no better than guessing. However this is infeasible since there are 2^{160} possible values of E .

It seems that the simplified scheme offers the same security as first proposal at the cost of smaller computation overhead.

However, this is not true as the simplified scheme can be attacked with lattice based fault cryptanalysis. This type of attack utilises errors in order to simplify the attack down to the problem of solving hidden number problem (HNP). HNP problem states that given pairs $\langle u_i, t_i^{(l)} \rangle$, where u_i is a random integer, $t_i^{(l)}$ denotes l subsequent bits of $t_i = b + au_i \text{ mod } q$, and a, b are constant (usually it is assumed that $t_i^{(l)}$ denotes most significant bits of t_i but this is not a requirement), one has to deduce the exact values of a and b [7]. As presented in [7], HNP

problem can be solved if $l > \epsilon\sqrt{\log q}$ for any fixed $\epsilon > 0$. Moreover, an algorithm that solves HNP may be used to attack the ElGamal-like signature scheme [7][8]. In such case a partial knowledge on k enables the attacker to recover the private key with less than 30 signatures. A similar attack can be also applied to the simplified modification of the DSA scheme.

To achieve this the attacker needs to induct errors into v used in the last step of the simplified signing procedure. Due to a such error the erroneous value \bar{s} equals

$$\bar{s} = k'(h(m) + v + e) \bmod q, \quad (24)$$

and may be used to guess the inducted error. This can be done by searching for e that satisfies

$$g^{s^{-1}h(m)}y^{s^{-1}r}g^{s^{-1}e} \bmod p \bmod q = r. \quad (25)$$

When proper e is found then the attacker gets partial information about v . Since $v = ar \bmod q$ and r is known, thus the attacker may collect sufficiently many data and solve the HNP problem for a .

Such an attack cannot be carried out in case of the previously proposed scheme (scheme [1]) since it computes v as

$$v = k + ar \bmod q, \quad (26)$$

and k is randomly chosen every iteration of the protocol. On the other hand, this modification requires additional additions to be performed in 4-th and 6-th step of signing.

Proposed modification of the DSA scheme is also immune to multiple fault attacks that can be inducted in practice [15]. Since there is no comparison procedure in our proposal, hence possible attack scenario may focus on inducting two errors: first error e into the private key a and the second error into one of the successive computations. Purpose of the second error is to mask error e during the inverse computation and thus force the device to output the erroneous signature that is suitable for fault cryptanalysis. Although this is possible scenario, it will be very difficult to achieve. This is due to error diffusion term T that depends on e and r which are unknown to the attacker during execution of the signing procedure. Therefore, probability that inducting multiple faults enables the attacker to break the proposed scheme is negligible.

7 Overhead of the Proposed Scheme

As presented in previous sections there is one additional (5) and three modified computations (3), (4), and (6) in the proposed modification of the DSA signature scheme (scheme [1]). The overhead of the modified computations is similar to the overhead of the original computations since only two additions modulo q and one EXOR operation is added. An additional computation of the error diffusion term T requires two exponentiations, one multiplication modulo p and q , and one addition modulo q . Accordingly, the computation overhead of the proposed modification is dominated by the time required to perform two exponentiations

and a multiplication. Therefore, the overhead of our proposal is smaller than the overhead of signature verification which requires three exponentiations and two multiplications modulo p and q .

Storage overhead of the proposed scheme is also higher compared to the storage required by the standard DSA scheme. It is so since our proposal utilises the public key y to verify the correctness of the private key used for signature generation. Therefore, the public key has to be stored inside of the device affecting storage requirements. However, in a real implementation this storage overhead can be neglected since cryptographic devices usually store the public key anyway.

Implementation overhead can be further reduced if we change the way error diffusion term T is computed. One of possible solutions is to limit the number of modular exponentiations. This can be achieved by using private key a instead of y , so that

$$T = (g^{-ar+v} \bmod p \bmod q) - r \bmod q. \quad (27)$$

This reduces the computational and storage overhead significantly but also affects the security of the whole proposal. In fact, it is now susceptible to attackers that can induct permanent errors or the same random error twice: first into a during computation of v , second during computation of T . In this way conducting a multiple fault attack enables to brake the proposal. Above mentioned flaw can be eliminated if we use $a^{-1} \bmod q$ instead of a and compute T as follows

$$T = (g^{a^{-1}(v-k)-r} \bmod p \bmod q) - 1 \bmod q. \quad (28)$$

Because multiplicative inverse is a nonlinear transformation thus the relation between error e , inducted into unknown private key a , and corresponding error affecting $a^{-1} \bmod q$ is unknown to the attacker. Therefore, probability that the multiple error attack succeeds is negligible. This is achieved at the cost of increased storage overhead which is required to store multiplicative inverse of the private key.

8 Conclusions

In this paper we have analysed the DSA scheme and its immunity to the fault cryptanalysis. We have demonstrated that introducing the error diffusion we can improve an immunity of the DSA scheme in the presence of faults affecting private key a . Our modification (scheme [1](#)) ensures that in order to recover the private key a the attacker needs to guess error E that depends both on inducted error e and randomly selected integer k (which is unknown to the attacker). Since E cannot be computed thus the attacker needs to check all of 2^{160} possible values of E for each e , which render the whole attack ineffective. Unlike simplified scheme (scheme [2](#)) the proposed modification is also immune to lattice-based attacks that take advantage of errors affecting v in the last step of signature generation.

Proposed modification also eliminates the comparison procedure which is an inherent part of signature verification and cause the obvious countermeasure to

be susceptible to multiple fault attacks. Since there is no such procedure in our proposal thus such attacks do not apply.

The overhead of the proposed scheme is similar to the overhead of the obvious countermeasure based on the signature verification. However, computational overhead can be further reduced if error diffusion term is calculated using inverse of the private key instead of the public key.

References

1. Anderson, R.J., Kuhn, M.G.: Tamper Resistance - a Cautionary Note. In: The Second USENIX Workshop on Electronic Commerce Proceedings, pp. 18–21 (1996)
2. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
3. Bao, F., Deng, R., Han, Y., Jeng, A., Narasimhalu, A.D., Ngair, T.-H.: Breaking Public Key Cryptosystems on Tamper Resistance Devices in the Presence of Transient Fault. In: Christianson, B., Lomas, M. (eds.) Security Protocols 1997. LNCS, vol. 1361, pp. 115–124. Springer, Heidelberg (1998)
4. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
5. Blmer, J., Otto, M., Seifert, J.-P.: A New CRT-RSA Algorithm Secure Against Bellcore Attacks. In: Proc. ACM Computer and Communications Security 2003 (ACM CCS 2003), pp. 311–320. ACM Press, New York (2003)
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
7. Boneh, D., Venkatesan, R.: Rounding in Lattices and Its Cryptographic Applications. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical, Experimental Analysis of Discrete Algorithms), pp. 675–681 (1997)
8. Breveglieri, L., Koren, I., Maistri, P., Ravasio, M.: Incorporating Error Detection in an RSA Architecture. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 71–79. Springer, Heidelberg (2006)
9. Czapski, M., Nikodem, M.: Error Correction Procedures for Advanced Encryption Standard. In: Int. Workshop on Coding and Cryptography (WCC 2007), April 16–20, 2007, pp. 89–98. INRIA (2007)
10. Dusart, P., Letourneux, G., Vivolo, O.: Differential Fault Analysis on A.E.S., ArXiv Computer Science e-prints (January 2003)
11. Giraud, C., Knudsen, E.: Fault Attacks on Signature Schemes. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 478–491. Springer, Heidelberg (2004)
12. Joye, M., Lenstra, A., Quisquater, J.J.: Chinese Remaindering Based Cryptosystems in the Presence of Faults. *Journal of Cryptology* 12, 241–245 (1999)
13. Karpovsky, M., Kulikowski, K.J., Taubin, A.: A Differential Fault Analysis Attack Resistant Architecture of the Advanced Encryption Standard. In: Proceedings of CARDIS 2004, pp. 177–192. Kluwer, Dordrecht (2004)
14. Karri, R., Wu, K., Mishra, P., Kim, Y.: Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Trans. on CAD of Integrated Circuits and Systems* 21(12), 1509–1517 (2002)

15. Kim, C.-H., Quisquater, J.-J.: Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 215–228. Springer, Heidelberg (2007)
16. Kömmerling, O., Kuhn, M.G.: Design Principles for Tamper-Resistant Smartcard Processors. In: USENIX Workshop on Smartcard Technology - Smartcard 1999, USENIX Association, pp. 9–20 (1999)
17. Naccache, D., Nguyen, P.Q., Tunstall, M., Whelan, C.: Experimenting with Faults, Lattices and the DSA. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 16–28. Springer, Heidelberg (2005)
18. Nguyen, P.Q., Shparlinski, I.E.: The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology* 15(3), 151–176 (2002)
19. Rosa, T.: Lattice-based Fault Attacks on DSA - Another Possible Strategy. In: Proceedings of the conference Security and Protection of Information 2005, Brno, Czech Republic, 3-5 May 2005, pp. 91–96 (2005)
20. Yen, S.M., Joye, M.: Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. *IEEE Transactions on Computers* 49(9), 967–970 (2000)
21. Yen, S.M., Kim, S., Lim, S., Moon, S.: RSA Speedup with Chinese Remainder Theorem Immune Against Hardware Fault Cryptanalysis. *IEEE Transactions on Computers* 52(4), 461–472 (2003)
22. Yen, S.M., Kim, D., Moon, S.: Cryptanalysis of Two Protocols for RSA with CRT Based on Fault Infection. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 53–61. Springer, Heidelberg (2006)

A Black Hen Lays White Eggs

Bipartite Multiplier Out of Montgomery One for On-Line RSA Verification

Masayuki Yoshino, Katsuyuki Okeya, and Camille Vuillaume

Hitachi, Ltd., Systems Development Laboratory, Kawasaki, Japan
{masayuki.yoshino.aa,katsuyuki.okeya.ue,camille.vuillaume.ch}@hitachi.com

Abstract. This paper proposes novel algorithms for computing double-size modular multiplications with few modulus-dependent precomputations. Low-end devices such as smartcards are usually equipped with hardware Montgomery multipliers. However, due to progresses of mathematical attacks, security institutions such as NIST have steadily demanded longer bit-lengths for public-key cryptography, making the multipliers quickly obsolete. In an attempt to extend the lifespan of such multipliers, double-size techniques compute modular multiplications with twice the bit-length of the multipliers. Techniques are known for extending the bit-length of classical Euclidean multipliers, of Montgomery multipliers and the combination thereof, namely bipartite multipliers. However, unlike classical and bipartite multiplications, Montgomery multiplications involve modulus-dependent precomputations, which amount to a large part of an RSA encryption or signature verification. The proposed double-size technique simulates double-size multiplications based on single-size Montgomery multipliers, and yet precomputations are essentially free: in an 2048-bit RSA encryption or signature verification with public exponent $e = 2^{16} + 1$, the proposal with a 1024-bit Montgomery multiplier is 1.4 times faster than the best previous technique.

Keywords: Montgomery multiplication, double-size technique, RSA, efficient implementation, smartcard.

1 Introduction

The algorithm proposed by Montgomery has been extensively implemented to perform costly modular multiplications which are time-critical for public-key cryptosystems such as RSA [Mon85, RSA78]. In particular, and unlike naive implementations of classical modular multiplications, Montgomery multiplications are not affected by carry propagation delays for computing the quotient of a modular reduction, and as a result, are suitable for high-performance hardware implementations. However, such accelerators are penalized by a strict restriction: their operand size is fixed. In order to deal with recent integer factoring records and ensure long-term security [Len04], official security institutions are updating their standards to longer key sizes than the mainstream 1024 bits for

RSA [Nis07, EMV, Ecr06]; unfortunately, such bit lengths are not supported by many cryptographic coprocessors.

This problem has motivated many studies for double-size modular multiplication techniques using single-size hardware modular multipliers. On the one hand, thanks to the Chinese Remainder Theorem, *private* operations (signature generation or decryption) can work with only single-size modular multiplications for computing double-size modular exponentiations [MOV96]. On the other hand, the Chinese Remainder Theorem is no help for *public* operations, and double-size techniques without using private keys are necessary. Following Paillier's seminal paper [Pai99], several solutions were proposed for simulating double-size classical modular multiplications with single-size classical modular multipliers [FS03, CJP03], and later, the techniques were adapted in order to simulate double-size Montgomery multiplications with the commonly used single-size Montgomery multiplier [YOV07a]. Finally, the less common but nonetheless promising bipartite multiplier [KT05], which includes a Montgomery and a classical multiplier working in parallel, was taking advantage of for simulating double-size bipartite multiplications [YOV07b].

In the context of public operations, RSA signature verification for instance, the verifier is unlikely to know the RSA modulus in advance; we refer to this event as *on-line* verification. On the one hand, classical modular multiplications are not affected by the fact that verification is performed off-line or on-line. With a bipartite multiplier, some modulus-dependent precomputations are required during on-line verification. However, when the parameters of the multiplier are appropriately chosen, the cost of precomputations is negligible [KT05]. But on the other hand, precomputations are far from being negligible when using Montgomery multipliers, especially when the public exponent is small. Assuming the 2048-bit exponentiation $X^e \bmod Z$, the basis X must be firstly converted to its Montgomery representation, namely $X * 2^{2048} \bmod Z$, which can be accomplished with 2048 successive shifts or eleven 2048-bit Montgomery multiplications; in the latter case, this amounts to 36% of the total verification time when $e = 2^{16} + 1$. This is especially unfortunate considering the fact that Montgomery multipliers represent the most popular architecture for cryptographic coprocessors [NM96].

In this paper, we solve the problem of costly on-line precomputations with a radically new approach. Although we assume a multiplier based on the celebrated Montgomery multiplication technique, we simulate a bipartite double-size multiplication, where on-line precomputations are essentially free. Although our double-size bipartite multiplication technique is slightly slower than double-size Montgomery multiplications, the penalty is largely counterbalanced by the benefit in terms of precomputations, at least when the public exponent e is small. When $e = 2^{16} + 1$, which is by far the most common choice for RSA, our technique is 1.4 times faster than the best previous techniques, and even more when $e = 3$. In addition, when the CPU and the coprocessor operate in parallel, which is possible on some low-cost microcontrollers, our proposal can be further optimized, leading to even greater speed. As a consequence, our simulated bipartite

multiplier is the fastest among double-size techniques for cryptographic devices equipped with Montgomery multipliers, and allows the current generation of such multipliers to comply with upcoming key-length standards of official institutes.

Notation: Let ℓ denote operand size of hardware modular multiplication units and L equal to 2ℓ . Small letters such as x , y and z denote ℓ -bit integers, and capital letters such as X , Y and Z denote L -bit integers, where Z is an odd modulus greater than 2^{L-1} like in the case of L -bit RSA.

2 Previous Double-Size Techniques

Montgomery multiplication algorithm has been extensively implemented as cryptographic coprocessors to help low-end devices performing heavy modular multiplications. However, the coprocessors are designed to support the main stream 1024-bit RSA, and face with the upper limit of their bit length to comply with upcoming key-length standards, such as the NIST recommendation; 2048-bit RSA. The problem has motivated double-size techniques to compute modular multiplication with twice the bit length of hardware multipliers.

2.1 Yoshino et al.'s Scheme

This subsection introduces Yoshino et al.'s work [YOVO7a, YOVO7b]: how to compute a double-size Montgomery multiplication with single-size Montgomery multiplications.

The double-size techniques proposed by Yoshino et al. require not only remainders but also quotients of single-size Montgomery multiplications. The equation $xy = q_m z + r_m c$ shows the relation among products of multiplier x and multiplicand y , quotient q_m and modulus z , and remainder r_m and constant c , where the constant c is usually selected as power of 2 for efficient hardware implementations in practice, therefore this paper also assumes such c satisfying $c = 2^\ell$ [MOV96].

Definition 1 shows an `mu` instruction for performing single-size Montgomery multiplications, outputting only the remainder.

Definition 1. For numbers, $0 \leq x, y < z$ and z is odd, the `mu` instruction is defined as $r_m \leftarrow \text{mu}(x, y, z)$ where $r_m \equiv xyc^{-1} \pmod{z}$.

Their double-size techniques assumed that an `mmu` instruction is available, which can be emulated with only 2 calls to single-size Montgomery multipliers, and computes the remainder r_m and the quotient q_m of Montgomery multiplications [YOVO7a] satisfying the equation $xy = q_m z + r_m c$.

Definition 2. For numbers, $0 \leq x, y < z$ and z is odd, the `mmu` instruction is defined as $(q_m, r_m) \leftarrow \text{mmu}(x, y, z)$ where $q_m = (xy - r_m c) / z$ and $r_m \equiv xyc^{-1} \pmod{z}$.

Yoshino et al.'s double-size techniques need two steps other than multiplier calls. First, every L -bit integer X , Y and Z is represented with ℓ -bit integers which can be handled by `mmu` instructions:

$$X = x_1(c - 1) + x_0c, Y = y_1(c - 1) + y_0c \text{ and } Z = z_1(c - 1) + z_0c.$$

Second, all quotients q_m and remainders r_m are sequentially gathered from mmu instructions.

Double-size Montgomery multiplications compute a remainder R_m such that $R_m \equiv XYC^{-1} \pmod{Z}$ where $0 \leq X, Y < Z$, and the constant C is called Montgomery constant, and twice bit length of the constant c : $C = 2^L (= c^2)$. Algorithm 1 shows their double-size Montgomery multiplications requiring 6 calls to mmu instructions, and 12 calls to Montgomery multipliers in total.

Algorithm 1. Double-size Montgomery multiplication [YOV07b](#)

INPUT: X, Y and Z where $0 \leq X, Y < Z$;
 OUTPUT: $XYC^{-1} \pmod{Z}$ where $C = 2^L$;

1. $(q_1, r_1) \leftarrow \text{mmu}(x_1, y_1, z_1)$
 2. $(q_2, r_2) \leftarrow \text{mmu}(q_1, z_0, c - 1)$ >// $c = 2^\ell$
 3. $(q_3, r_3) \leftarrow \text{mmu}(x_0 + x_1, y_0 + y_1, c - 1)$
 4. $(q_4, r_4) \leftarrow \text{mmu}(x_0, y_0, c - 1)$
 5. $(q_5, r_5) \leftarrow \text{mmu}(c - 1, -q_2 + q_3 - q_4 + r_1, z_1)$
 6. $(q_6, r_6) \leftarrow \text{mmu}(q_5, z_0, c - 1)$
 7. **return** $(q_2 + q_4 - q_6 - r_1 - r_2 + r_3 - r_4 + r_5)(c - 1) + (r_2 + r_4 - r_6)c \pmod{Z}$
-

Thanks to Algorithm 1, one can set a new MU instruction to compute L -bit Montgomery multiplications such that $R_m \leftarrow \text{MU}(X, Y, Z)$ where $R_m \equiv XYC^{-1} \pmod{Z}$, $0 \leq X, Y < Z$ and $C = 2^L$.

2.2 L -Bit RSA Public Operations

The MU instruction (double-size Montgomery multiplications) introduced in last subsection requires twelve single-size multiplications and other basic modular operations; therefore the number of calls to the MU instruction should be as small as possible to get better performance. This subsection explains the contributions and weak points of previous double-size techniques to RSA public operations, which is the most popular application for double-size techniques.

L -bit RSA public operations (signature verification and encryption) employ an L -bit modular exponentiation with a small exponent, following that $X^e \pmod{Z}$, where the ciphertext or signature X , the public modulus Z , and a small public exponent e . The binary method commonly used for RSA public operations computes double-size Montgomery multiplications and squarings according to the bit pattern of the public exponent e . Algorithm 2 shows a left-to-right binary method, which scans e from the most significant bit e_k to the least significant bit e_0 bit-by-bit.

From the view of efficient computation and mathematical security, the exponent used for RSA *public* operations is much smaller than for *private* operations [MOV96](#), [RSA95](#). Currently, by far the most common value of the

Algorithm 2. Binary method from the most significant bit

INPUT: X, Z and small public exponent $e = (e_k \cdots e_i \cdots e_0)_2$ where $0 \leq X < Z$;
 OUTPUT: $X^e \pmod{Z}$;

1. $Y \leftarrow C^2 \pmod{Z}$ // $C = 2^L$
 2. $T \leftarrow \text{MU}(X, Y, Z)$
 3. $Y \leftarrow T$
 4. **for** i **from** $k - 1$ **down to** 0 **do**
 - (a) $T \leftarrow \text{MU}(T, T, Z)$ //squaring
 - (b) **if** $e_i = 1$, **do**
 - i. **if** $i \neq 0$ **then** $T \leftarrow \text{MU}(T, Y, Z)$ //multiplication
 - ii. **if** $i = 0$ **then** $T \leftarrow \text{MU}(T, X, Z)$ //multiplication and reduction
 5. **return** T
-

public exponent e is $2^{16} + 1$ having only two 1's in its binary representation ($= (1000000000000001)_2$). In the case of public exponent $e = 2^{16} + 1$, MU instruction is called only 18 times from Step 2 to Step 5 of Algorithm 2. In addition to that, the Algorithm 2 Step 1 seems to be quite cheap, however, this simple modular squaring is seriously expensive for double-size RSA public operations, as it will be explained below.

2.3 Previous Approaches for On-Line Precomputations

There are important differences between *private* and *public* operations: **off-line** precomputations are possible in the former case whereas the latter case requires **on-line** precomputations.

On-line precomputations in Algorithm 2; Step 1 consists of a simple L -bit modular squaring which might look cheap at first sight; however this is not true for low-end devices such as smartcards. There are two known approaches with/without help from Montgomery multipliers; unfortunately, both are seriously slow, and damage performances of double-size techniques on low-end devices.

(1) Approach with MU Instruction:

In an attempt to benefit from hardware accelerators, Algorithm 3 employs MU instructions to perform a L -bit modular squaring ($C^2 \pmod{Z}$) using the binary method. Thanks to the cryptographic coprocessor, the approach looks fast, but in fact, the calculation costs are quite heavy: in the case of a 2048-bit modular squaring, Algorithm 3 takes 120 calls to the multiplier, since MU instruction requires 12 calls to the multiplier and is called 10 times by the binary method. As a consequence, the approach with the MU instruction is very costly considering that it only computes a simple modular squaring.

(2) CPU approach:

Theoretically, the CPU can compute any-bit modular multiplications without help from hardware accelerators including the L -bit modular squaring

Algorithm 3. L -bit modular squaring ($C^2 \pmod{Z}$) with MU instructions

INPUT: bitlength $L = (L_{L-1} \cdots L_\ell \cdots L_0)_2$ and modulus Z ;

OUTPUT: $C^2 \pmod{Z}$ where $C = 2^L$;

1. $D \leftarrow 2C \pmod{Z}$ **and** $T \leftarrow 2C \pmod{Z}$
 2. **for** i **from** $\lfloor \log_2 L \rfloor - 2$ **down to** 0 **do**
 - (a) $D \leftarrow \text{MU}(D, D, Z)$
 - (b) **if** $L_i = 1$ **then** $D \leftarrow \text{MU}(D, T, Z)$
 3. **return** D
-

($C^2 \pmod{Z}$). The approach of Algorithm 4 is taken by computers whose CPUs are powerful enough not to need help from hardware accelerators, however, this is not the case for the low-end devices where the performance gap between CPU and arithmetic coprocessor is usually quite large. As a result, Algorithm 4 is practically much slower than Algorithm 3 in these environments.

Algorithm 4. L -bit modular squaring with only CPU instructions

INPUT: bitlength $L = (L_{L-1} \cdots L_\ell \cdots L_0)_2$ and modulus Z ;

OUTPUT: $C^2 \pmod{Z}$ where $C = 2^L$;

1. $D \leftarrow C - Z$
 2. **for** i **from** $\ell - 1$ **down to** 0 **do**
 - (a) $D \leftarrow 2D$
 - (b) **if** $D \geq C$, **then** $D \leftarrow D - Z$.
 3. **if** $D \geq Z$, **then** $D \leftarrow D - Z$.
 4. **return** D
-

3 New Double-Size Bipartite Multiplication

L -bit RSA *public* operations require a simple but expensive **on-line** modular-dependent precomputation for low-end devices with ℓ -bit Montgomery multipliers. This section presents new double-size techniques for such environments, which derive their high performance from Montgomery multipliers while eliminating almost all precomputations.

3.1 Overview

The proposal mixes two different modular multiplication algorithms which are executable with the usual Montgomery multipliers. Fig. 1 shows a design of our techniques: first, L -bit integers X , Y and Z are divided into ℓ -bit integers, and inputted to a hardware accelerator outputting the ℓ -bit remainder r_m of Montgomery multiplications. In addition to single-size *Montgomery* multiplications, the new techniques employ single-size *classical* multiplications. Second, their remainders (r_m and r_c) and quotients (q_m and q_c) are computed based on only

the remainder r_m . Last, the remainders and quotients are assembled to build a double-size remainder R satisfying

$$R \equiv XYc^{-1} \pmod{Z},$$

where $0 \leq X, Y < Z$. The new modular multiplication is accompanied by the constant c , which is only half the bit length of the Montgomery constant C , contributing to the fact that our new on-line precomputations can be performed at much cheaper cost.

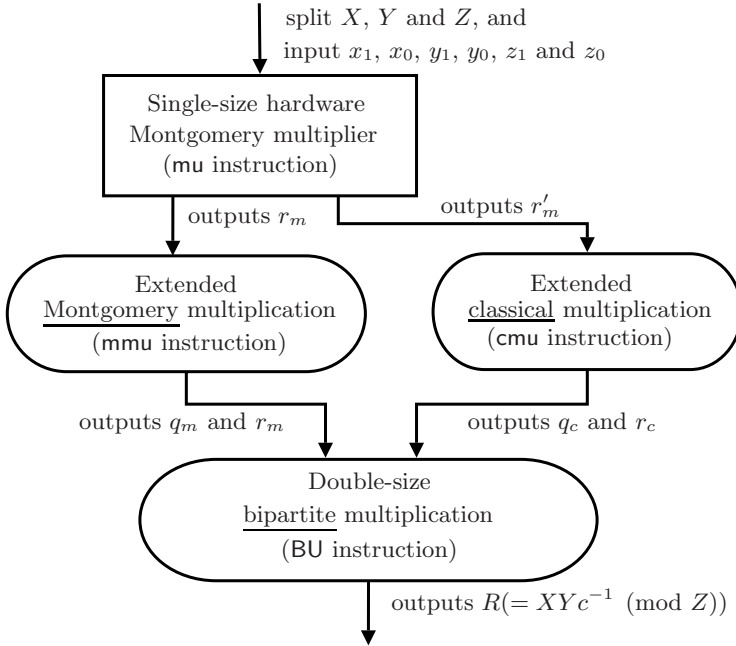


Fig. 1. Configuration of New Double-Size Bipartite Multiplication

3.2 How to Divide L -Bit Integers for the ℓ -Bit Multiplier

In order to benefit from hardware accelerators which can handle only ℓ -bit arithmetic operations, L -bit integers can be simply divided into upper and lower ℓ bits such that $X = x_1c + x_0$, where x_1 is upper and x_0 is lower ℓ bits of X . However, Montgomery multiplications require *odd* modulus¹. In order to prepare odd moduli, Algorithm 5 derives from the following equation: $Z = z_1c + z_0 = (z_1 + 1)c - (c - z_0)$.

¹ In fact, it is possible to perform Montgomery multiplications with *even* modulus [Koc94]. However, the technique requires other arithmetic operations in addition to the multiplications in hardware: this costly technique is not considered in our paper.

Algorithm 5. L -bit modulus division with odd ℓ -bit moduli

INPUT: odd Modulus Z ;

OUTPUT: odd moduli z_1 and z_0 such that $Z = z_1c + z_0$ with $c = 2^\ell$;

1. $z_1 \leftarrow \lfloor Z/c \rfloor$ **and** $z_0 \leftarrow Z \pmod{c}$.
 2. **if** z_1 **is even**, $z_1 \leftarrow z_1 + 1$ **and** $z_0 \leftarrow z_0 - c$.
 3. **return** (z_1, z_0)
-

3.3 New ℓ -Bit Instructions Based on an ℓ -Bit Multiplier

This subsection defines new instructions to output quotients and remainders of classical multiplications and Montgomery multiplications, which can be built on the usual Montgomery multiplier.

Similar with Definition 2 in Section 2.1, the equation; $xy = q_cz + r_c$ shows the relation between the remainder r_c and the quotient q_c of *classical* multiplications, which can be implemented with only three calls to the mu instruction.

Definition 3. For numbers, $0 \leq x, y < z$ and z is odd, the cmu instruction is defined as $(q_c, r_c) \leftarrow \text{cmu}(x, y, z)$ where $q_c = (xy - r_c)/z$ and $r_c \equiv xy \pmod{z}$.

Algorithm 6 shows how to simulate the cmu instruction with the mu instruction; and the correctness is proven in Appendix A.1.

Algorithm 6. The cmu Instruction based on The mu Instruction

INPUT: x, y, z and t with $0 \leq x, y < z$, z is odd and $t = c^2 \pmod{z}$;

OUTPUT: q_c and r_c , where $q_c = (xy - r_c)/z$ and $r_c \equiv xy \pmod{z}$;

1. $x' \leftarrow \text{mu}(x, t, z)$ // $\equiv xc \pmod{z}$
 2. $r_c \leftarrow \text{mu}(x', y, z)$ // $\equiv xy \pmod{z}$
 3. $r'_c \leftarrow \text{mu}(x', y, z + 2)$ // $\equiv xy \pmod{z + 2}$
 4. $q_c \leftarrow (r_c - r'_c)$
 5. (a) **if** q_c **is odd**, **then** $q_c \leftarrow (q_c + z + 2)/2$
 (b) **else if** q_c **is even and negative**, **then** $q_c \leftarrow q_c/2 + z + 2$
 6. **return** (q_c, r_c)
-

3.4 How to Build an L -Bit Remainder with ℓ -Bit Instructions

Finally, this subsection presents how to build a remainder of new double-size modular multiplication on the remainders and the quotients of single-size modular multiplications.

Definition 4 shows the BU instruction for computing L -bit *bipartite* multiplication.

Definition 4. For numbers, $0 \leq X, Y < Z$, the BU instruction is defined as $R \leftarrow \text{BU}(X, Y, Z)$ where $R \equiv XYc^{-1} \pmod{Z}$ and $c = 2^\ell$.

The BU instruction performs L -bit modular multiplication; $XYc^{-1} \pmod{Z}$ accompanied with only the ℓ -bit constant c , which is only half the size of the Montgomery constant C , contributing to the fact that our new precomputations can be performed at much cheaper cost.

Algorithm 7 shows how to use the mmu instruction and the cmu instruction to build the BU instruction; the correctness is proven in Appendix A.2.

Algorithm 7. The BU Instruction based on The mmu and cmu Instructions

INPUT: X, Y and Z , where $X = x_1c + x_0, Y = y_1c + y_0$ and $Z = z_1c + z_0$;
 OUTPUT: $XYc^{-1} \pmod{Z}$;

1. $(q_1, r_1) \leftarrow \text{cmu}(x_1, y_1, z_1)$
 2. $(q_2, r_2) \leftarrow \text{mmu}(q_1, z_0, c - 1)$
 3. $(q_3, r_3) \leftarrow \text{mmu}(x_0, y_0, c - 1)$
 4. (a) **if** z_0 **is positive**, $(q_4, r_4) \leftarrow \text{mmu}(q_2 + q_3, c - 1, z_0)$
 (b) **if** z_0 **is not positive**, $(q_4, r_4) \leftarrow \text{mmu}(-q_2 + q_3, c - 1, z_0)$
 5. $(q_5, r_5) \leftarrow \text{mmu}(q_4, z_1, c - 1)$
 6. $(q_6, r_6) \leftarrow \text{mmu}(x_1 + x_0, y_1 + y_0, c - 1)$
 7. (a) **if** z_0 **is positive**,
 $R \leftarrow (-r_1 + r_2 + r_3 + r_4 + q_2 + q_3 + q_5 - q_6) + (r_1 - r_2 - r_3 - r_5 + r_6 - q_2 - q_3 - q_5 + q_6)c$
 (b) **if** z_0 **is not positive**,
 $R \leftarrow (-r_1 - r_2 - r_3 + r_4 - q_2 + q_3 - q_5 - q_6) + (r_1 + r_2 + r_3 + r_5 + r_6 + q_2 - q_3 + q_5 + q_6)c$
 8. **return** $R \pmod{Z}$
-

Since the cmu instruction based on Algorithm 6 requiring three calls to the Montgomery multiplier is costlier than the mmu instruction requiring only two calls [YOVO7a], Algorithm 7 minimizes calls to the cmu instruction, which appears only once in Step 1.

Fig. 2 shows a design of our double-size modular multiplication: the first three lines show components of the product XY , which is computed by the

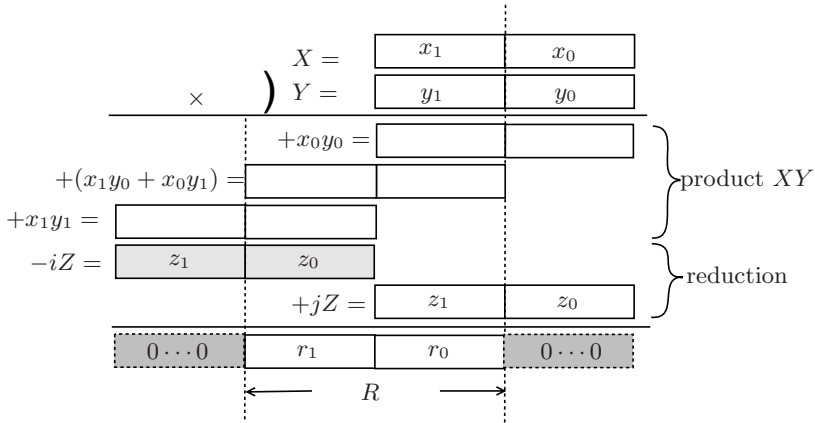


Fig. 2. Sketch of Double-Size Bipartite Multiplications

mmu instruction. There are two kinds of modular reductions in the other steps: One is to subtract Z from the most significant (left) side, which is based on the cmu instruction, and the other adds Z from the least significant (right) side based on the mmu instruction. Finally, one can discard each ℓ -bit integer from the most and least significant side, and get the L -bit remainder located in the middle.

4 Evaluation

This section shows how the proposal speeds up on-line precomputations. NIST recommends using 2048-bit RSA instead of the mainstream 1024-bit RSA from 2010 though 2030 [Nis07]; this paper follows the NIST recommendation, and evaluates the proposed techniques with 2048-bit RSA on smartcards which can only handle 1024-bit Montgomery multiplications.

4.1 Few On-Line Precomputations

L -bit RSA public operations consist of an L -bit modular exponentiation: $X^e \pmod{Z}$, with the ciphertext or signature X , the public modulus Z , and the small public exponent e . The RSA public operations can be performed by Algorithm 8, which is a left-to-right binary method with the BU instruction presented in Definition 4, and looks similar to Algorithm 2 with the MU instruction requiring heavy precomputations. However, a precomputation of Algorithm 8 (Step 1) is essentially free thanks to the following equation: $c^2 \pmod{Z} = C - Z$, where $c^2 = C = 2^L$ and $2^{L-1} < Z < 2^L$.

Algorithm 8. Binary method from the most significant bit based on BU instruction

INPUT: X, Z and small public exponent $e = (e_k \cdots e_i \cdots e_0)_2$ where $0 \leq X < Z$;

OUTPUT: $X^e \pmod{Z}$;

1. $Y \leftarrow c^2 \pmod{Z}$ // = $C - Z$
 2. $T \leftarrow \text{BU}(X, Y, Z)$
 3. $Y \leftarrow T$
 4. **for** i **from** $k - 1$ **down to** 0 **do**
 - (a) $T \leftarrow \text{BU}(T, T, Z)$ //squaring
 - (b) **if** $e_i = 1$, **do**
 - i. **if** $i \neq 0$ **then** $T \leftarrow \text{BU}(T, Y, Z)$ //multiplication
 - ii. **if** $i = 0$ **then** $T \leftarrow \text{BU}(T, X, Z)$ //multiplication and reduction
 5. **return** T
-

The BU instruction requires other on-line precomputation $c^2 \pmod{z}$ for cmu instruction, which is called at Algorithm 7 Step 1. This precomputation can easily be performed using Algorithm 9 with only several calls to the hardware multiplier.

Algorithm 9. ℓ -bit modular squaring with mu instructionsINPUT: bitlength $\ell = (\ell_{\ell-1} \cdots \ell_i \cdots \ell_0)_2$ and modulus z ;OUTPUT: $c^2 \pmod{z}$ where $c = 2^\ell$;

1. $d \leftarrow 2c \pmod{z}$ **and** $t \leftarrow 2c \pmod{z}$
2. **for** i **from** $\lfloor \log_2 \ell \rfloor - 2$ **down to** 0 **do**
 - (a) $d \leftarrow \text{mu}(d, d, z)$
 - (b) **if** $\ell_i = 1$ **then** $d \leftarrow \text{mu}(d, t, z)$
3. **return** d

4.2 Performance Improvement

The proposed double-size techniques are evaluated for smartcards which can only handle 1024-bit Montgomery multiplications in the case of 2048-bit RSA public operations with the common exponent $e = 2^{16} + 1$ to follow the NIST recommendation [Nis07]. Table 1 includes the performance of the 2048-bit RSA with three columns; on-line precomputations, a modular exponentiation and the total, which are evaluated by the binary (square-and-multiply) methods following Algorithm 2 or Algorithm 8.

The proposal eliminates almost all on-line precomputations, and contributes to improve the total performance: One of the on-line precomputation; $C \pmod{Z}$ is replaced with a subtraction; $C - Z$, and the other precomputation; $c^2 \pmod{z}$ requires only 9 calls to the Montgomery multipliers, therefore the proposal advantages in the on-line precomputations. As a result, the proposed method costs only 70% ($\simeq 243/336$) of the best previous method.

Figure 3 depicts how the cost, expressed in number of calls to the single-size Montgomery multiplier, varies with the exponent e in the case of a 2048-bit RSA encryption. For exponents of less than 32 bits, our proposal is always better than previous techniques. The turnover when double-size Montgomery multiplications [YOV07a] becomes more competitive than our proposal occurs for the 65-bit exponent $e = (11 \dots 11)_2$. However, we argue that in practice, small RSA exponents of less than 32 bits represent the overwhelming majority of cases [RSA95].

4.3 Further Performance Improvement

Some micro processor can perform modular operations in parallel with help of cryptographic coprocessors and CPU: while the coprocessors work, CPU can compute other arithmetic modular operations. Despite gap between the speed

Table 1. Calls to the multiplier in the 2048-bit RSA public operation

Scheme	On-line precomputations	Modular exponentiation	Total
[YOV07a]	140	252	392
[YOV07b]	120	216	336
This paper	9	234	243

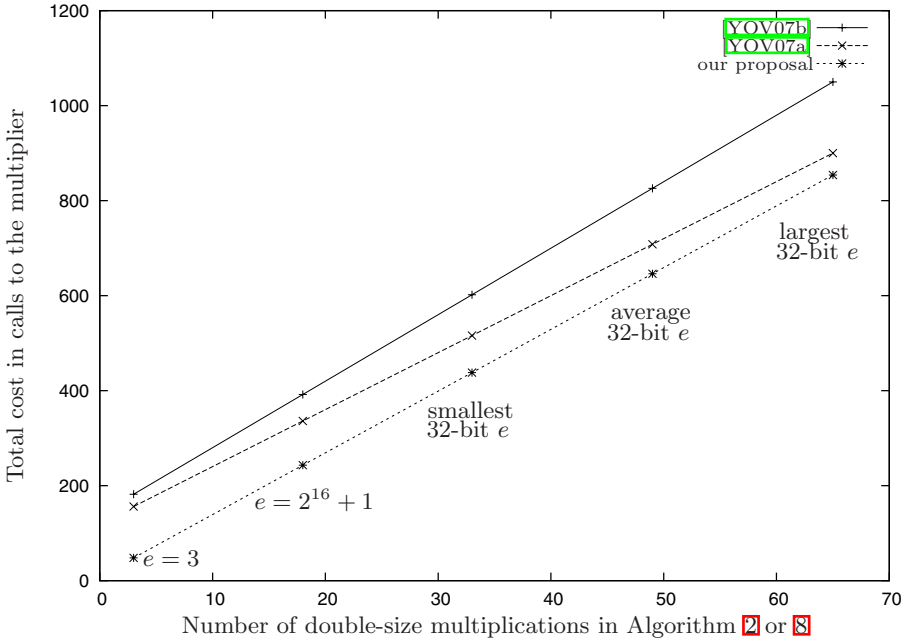


Fig. 3. Calls to the Montgomery multiplier for several exponents e

of those processors, such environments can accelerate double-size modular multiplication assigning some step of Algorithm 8 in the arithmetic processor and the other steps in CPU such as Step 1–5 in the coprocessor and Step 6 in CPU, or Step 1,2,4,5 in the coprocessor and the others in CPU. Therefore, parallel operations help to optimize our proposal, leading to even greater speed.

5 Conclusion

This paper proposed novel double-size modular multiplication algorithms with few modulus-dependent precomputations for on-line RSA public operations, which gave birth to double-size bipartite multiplication on the most commonly used single-size Montgomery multipliers in order to eliminate heavy precomputations required by all previous double-size Montgomery multiplication techniques. Although the proposed double-size bipartite multiplication technique is slightly slower than the best technique of double-size Montgomery multiplication, the penalty is largely counterbalanced by the benefit in terms of precomputations: when the public exponent is $e = 2^{16} + 1$, which is by far the most common choice for RSA, our method is 1.4 times faster than the best previous techniques. In addition, when the CPU and the coprocessor operate in parallel, which is possible for some low-cost micro controllers, our proposal can be further optimized, leading to even greater speed. As a consequence, our double-size bipartite multiplication technique is the fastest among all double-size techniques for the cryptographic devices equipped with hardware Montgomery multipliers.

References

- [Koc94] Koç, Ç.K.: Montgomery Reduction with Even Modulus. IEE Proceedings - Computers and Digital Techniques 141(5), 314–316 (1994)
- [CJP03] Chevallier-Mames, B., Joye, M., Paillier, P.: Faster Double-Size Modular Multiplication From Euclidean Multipliers. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 214–227. Springer, Heidelberg (2003)
- [Ecr06] European Network of Excellence in Cryptology (ECRYPT). ECRYPT Yearly Report on Algorithms and Keysizes (2006), <http://www.ecrypt.eu.org/documents/D.SPA.21-1.1.pdf>
- [EMV] EMV. EMV Issuer and Application Security Guidelines, Version 2.1 (2007), <http://www.emvco.com/specifications.asp?show=4>
- [FS03] Fischer, W., Seifert, J.-P.: Increasing the Bitlength of Crypto-coprocessors. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 71–81. Springer, Heidelberg (2003)
- [KT05] Kaihara, M.E., Takagi, N.: Bipartite modular multiplication. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 201–210. Springer, Heidelberg (2005)
- [Len04] Arjen, K.: Lenstra. Key Lengths (2004), http://cm.bell-labs.com/who/akl/key_lengths.pdf
- [Mon85] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
- [MOV96] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
- [Nis07] National Institute of Standards and Technology. NIST Special Publication 800-57 Recommendation for Key Management Part 1: General (Revised) (2007), <http://csrc.nist.gov/CryptoToolkit/tkkeygmt.html>
- [NM96] Naccache, D., M'Raihi, D.: Arithmetic co-processors for public-key cryptography: The state of the art. In: CARDIS, pp. 18–20 (1996)
- [Pai99] Paillier, P.: Low-Cost Double-Size Modular Exponentiation or How to Stretch Your Cryptoprocessor. In: Imai, H., Zheng, Y. (eds.) PKC 1999. LNCS, vol. 1560, pp. 223–234. Springer, Heidelberg (1999)
- [RSA78] Rivest, R.L., Shamir, A., Adelman, L.M.: A Method for Obtaining Digital Signatures and Public-key Cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
- [RSA95] RSA Laboratories. The Secure Use of RSA. CryptoBytes 1(3) (1995), <ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto1n3.pdf>
- [YOV07a] Yoshino, M., Okeya, K., Vuillaume, C.: Unbridle the Bit-Length of a Crypto-Coprocessor with Montgomery Multiplication. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 188–202. Springer, Heidelberg (2007)
- [YOV07b] Yoshino, M., Okeya, K., Vuillaume, C.: Double-Size Bipartite Modular Multiplication. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP 2007. LNCS, vol. 4586, pp. 230–244. Springer, Heidelberg (2007)

A Proof for Correctness

A.1 Algorithm 6: The cmu Instruction Based on a mu Instruction

Algorithm of Montgomery multiplications is different from classical multiplications; however, one can simulate classical multiplications easily using the ℓ -bit mu instruction implementing Montgomery multiplications thanks to the following equation:

$$r_c = xy \pmod{z} = x'yc^{-1} \pmod{z}$$

where $0 \leq x, y < z$ and $x' = xc \pmod{z}$. Therefore, the mu instruction can output the classical remainder r_c according to the following two intuitive steps:

1. $x' \leftarrow \text{mu}(x, c^2 \pmod{z})$
2. $r_c \leftarrow \text{mu}(x', y, z)$

Thanks to these steps, one can compute r_c with help from the multipliers.

There is a requirement for Montgomery multiplications: only odd moduli are available. The following proof show how to compute classical quotient $q_c = (xy - r_c)/z$ from two different classical remainders.

Proof. For numbers, where $0 \leq x, y < z$ and z is odd, classical multiplication outputs a quotient q_c and a remainder r_c , which satisfy the following equation: $xy = q_cz + r_c$ where $q_c = (xy - r_c)/z$ and $r_c \equiv xy \pmod{z}$. Equivalently,

$$\begin{aligned} xy &= q_cz + r_c \\ &= q_c(z + 2) + (-2q_c + r_c) \end{aligned} \tag{1}$$

$$= q'_c(z + 2) + r'_c \tag{2}$$

From the equation (1) and (2),

$$q_c = (r_c - r'_c + \delta(z + 2))/2$$

holds with some integer δ .

Since $0 \leq x, y < z$ holds, q_c , r_c and r'_c satisfy the following conditions: $0 \leq q_c < z$, $0 \leq r_c < z$ and $0 \leq r'_c < z + 2$. From the equation $-(z + 2) < (r_c - r'_c) < z$, the following condition holds:

$$\text{If value of } (r_c - r'_c) \text{ is } \begin{cases} \text{even and non negative, then} & \delta = 0 \\ \text{odd, then} & \delta = 1 \\ \text{even and negative, then} & \delta = 2 \end{cases} \quad \square$$

A.2 Algorithm 7: The BU Instruction Based on a mmu and cmu Instruction

Algorithm 7 builds the BU instruction on a cmu instruction and an mmu instruction, and needs to process branches in Step 4 and Step 7 whether z_0 is positive or not. This paper only introduces a proof in the case that z_0 is positive, but one can follow the other case similarly.

ⁱ Algorithm 9 can help to precompute the equation $c^2 \pmod{z}$.

Proof. L -bit modulus Z is represented by Algorithm 5 as the followings:

$$Z = z_1c + z_0$$

where $0 < z_1 < c$ and $-c < z_0 < c$, and the other L -bit integers X and Y are simply divided into upper and lower ℓ -bit integers.

$$X = x_1c + x_0 \text{ and } Y = y_1c + y_0.$$

where $0 \leq x_1, x_0, y_1, y_0 < c$. Then, the following equation holds.

$$XY = x_1y_1c(c-1) + (x_1 + x_0)(y_1 + y_0)c - x_0y_0(c-1) \quad (3)$$

The first term of Equation (3) is transformed into the following equations with the first call to a `cmu` instruction and the second call to an `mmu` instruction.

$$\begin{aligned} x_1y_1c(c-1) &= (q_1z_1 + r_1)c(c-1) \\ &\equiv (-q_1z_0 + r_1c)(c-1) \quad (\because z_1c \equiv -z_0 \pmod{Z}) \\ &= (q_2 + (r_1 - r_2 - q_2)c)(c-1) \end{aligned}$$

The third term of Equation (3) is also transformed into the following equation with a call to the `mu` instruction.

$$x_0y_0(c-1) = (-q_3 + (r_3 + q_3)c)(c-1)$$

Therefore, the first and third term of Equation (3) are combined with twice the help from the `mmu` instruction.

$$\begin{aligned} x_1y_1c(c-1) + x_0y_0(c-1) &= (q_2 + q_3)(c-1) + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \\ &= (q_4z_0 + r_4c) + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \\ &\equiv (-q_4z_1 + r_4c) + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \quad (\because z_0 \equiv -z_1c \pmod{Z}) \\ &= ((q_5 + r_4) - (q_5 + r_5)c)c + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \end{aligned}$$

The second term of Equation (3) is transformed into the followings with last call to the `mu` instruction.

$$(x_1 + x_0)(y_1 + y_0)c = (-q_6 + (r_6 + q_6)c)c$$

Finally, Equation (3) consisting of three terms are concluded with the following equations.

$$\begin{aligned} XY \equiv & (-r_1 + r_2 + r_3 + r_4 + q_2 + q_3 + q_5 - q_6) \\ & + (r_1 - r_2 - r_3 - r_5 + r_6 - q_2 - q_3 - q_5 + q_6)c \pmod{Z}. \quad \square \end{aligned}$$

Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents

Carsten Rolfes, Axel Poschmann, Gregor Leander,
and Christof Paar

Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{rolfes, poschmann, cpaar}@crypto.rub.de,
leander@itsc.rub.de

Abstract. In recent years more and more security sensitive applications use passive smart devices such as contactless smart cards and RFID tags. Cost constraints imply a small hardware footprint of all components of a smart device. One particular problem of all *passive* smart devices such as RFID tags and contactless smart cards are the harsh power constraints. On the other hand, *active* smart devices have to minimize *energy* consumption. Recently, many lightweight block ciphers have been published. In this paper we present three different architecture of the ultra-lightweight algorithm PRESENT and highlight their suitability for both active and passive smart devices. Our implementation results of the serialized architecture require only 1000 GE. To the best of our knowledge this is the smallest hardware implementation of a cryptographic algorithm with a moderate security level.

1 Background

Smart cards are widely in use for authentication, access control, and payment purposes. Their applications range from access control of ski resorts and soccer stadiums over parking lots to highly secured areas of both company and government buildings. MasterCard, Visa, and JCB are currently defining specifications for contact and contactless payments using smart cards in their EMV standards [17][16]. In recent years there has been an increasing trend towards contactless smart cards. In fact, contactless smart cards are a special subset of passive RFID tags [8]. With regards to the terminology of pervasive computing both can be summarised by the term *passive smart devices*. Even for very sensitive data passive smart devices are used, e.g. Visa payWave card [22], hence, security mechanism play a key role for these applications.

Many smart devices, especially commodities, are very cost sensitive. If the volumes are large enough –and this is indicated by the term *pervasive*– an application specific integrated circuit (ASIC) will nearly always be cheaper than a programmable micro computer. In hardware the price of an ASIC is roughly equivalent to the area in silicon it requires. The area is usually measured in μm^2 , but this value depends on the fabrication technology and the standard cell

library. In order to compare the area requirements independently it is common to state the area as *gate equivalents* (GE). One GE is equivalent to the area which is required by the two-input NAND gate with the lowest driving strength of the corresponding technology. The area in GE is derived by dividing the area in μm^2 by the area of a two-input NAND gate.

Moreover, one particular problem of all *passive* smart devices such as RFID tags and contactless smart cards are the harsh *power* constraints. Depending on the transmission range of the application (close, proximity, or vicinity coupling), the power constraints can be as strict as a few μW . Therefore, the ASIC and all of its components have to be designed with special care for the total *power* consumption. *Active* smart devices, such as wireless sensor nodes, RFID reader handhelds, or contact smart cards have their own power supply, i.e. a battery, or are powered by the reading device via a physical contact. Therefore, the power constraints are more relaxed for this device class. The main design goal here is to minimize the total *energy* consumption and the overall execution time.

Block ciphers are the working horses of the cryptographic primitives. Unfortunately, a vast majority of block ciphers have been developed with good software properties in mind, which in turn means that the gate count for a hardware implementation is rather high. In order to cope with this situation quite a few cryptographic algorithms have been published that are especially optimized for ultra-constrained devices. Examples for lightweight stream ciphers are Grain and Trivium and examples for lightweight block ciphers are DESXL [13], HIGHT [5], mCrypton [14], PRESENT [3], and SEA [15]. This research area is also referred to as *low cost* or *lightweight cryptography*. Some designers kept the algorithm secret in order to gain additional security by obscurity. However, the cryptanalyses of two widely used lightweight algorithms show that this violation of the *Kerckhoff principle* [12] is prohibitive: *Keeloq* [1] and *Mifare* both were broken shortly after their algorithm was reverse-engineered [2,18].

PRESENT is an aggressively hardware optimized ultra-lightweight block cipher, first presented at CHES 2007 [3]. According to the authors PRESENT was developed with a minimal hardware footprint (1570 GE) in mind such that it is suitable for passive RFID tags. However, in this work we show that a serialized implementation can be realized with as few as 1000 GE, which make it especially interesting for all kind of low cost passive smart devices. Moreover we propose two additional architectures which are suitable for low cost and high end active smart devices.

In the remainder of this work, we first recall the specification of PRESENT in Section 2. We propose three different hardware architectures of PRESENT in Section 3 and the implementation results are evaluated in Section 4. Finally, in Section 5 we conclude the paper.

2 The PRESENT Algorithm

PRESENT is a substitution-permutation network with 64-bits block size and 80 or 128 bits of key (from here on referred to as PRESENT for the 80 bit version

```

generateRoundKeys()
for i = 1 to 31 do
    addRoundKey(STATE, Ki)
    sBoxLayer(STATE)
    pLayer(STATE)
end for
addRoundKey(STATE, K32)
    
```

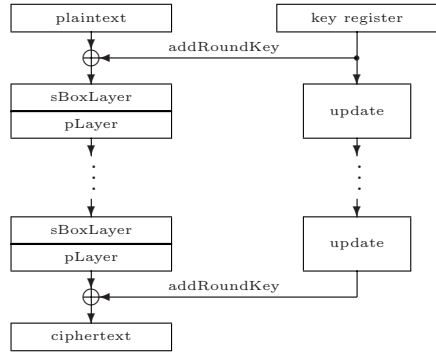


Fig. 1. A top-level algorithmic description of PRESENT

and PRESENT-128 for the 128 bit version). In the remainder of this article we focus on PRESENT, because 80-bits provide a security level which is sufficient for many RFID driven applications. PRESENT has 31 regular rounds and a final round that only consists of the key mixing step. One regular round consists of a key mixing step, a substitution layer, and a permutation layer.

The substitution layer consists of 16 S-Boxes in parallel that each have 4 bit input and 4 bit output (4×4): $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. The S-Box is given in hexadecimal notation according to the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

The bit permutation used in PRESENT is given by the following table. Bit i of STATE is moved to bit $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

The key schedule of PRESENT consists of a 61-bit left rotation, an S-Box, and an XOR with a round counter. Note that PRESENT uses the same S-Box for the datapath and the key schedule, which allows to share resources. The user-supplied key is stored in a key register and its 64 most significant (i.e. leftmost) bits serve as the round key. The key register is rotated by 61 bit positions to the left, the left-most four bits are passed through the PRESENT S-Box, and the `round_counter` value i is exclusive-ored with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of K with the least significant bit of `round_counter` on the right. Figure 1 provides a top-level

description of the PRESENT algorithm. For further details, the interested reader is referred to [3].

3 Three Different Architectures of PRESENT Implementations

For different application scenarios there exists also different demands on the implementation and the optimization goals. An implementation for a low cost passive smart device, such as RFID tags or contactless smart cards requires small area and power consumption, while the throughput is of secondary interest. On the other hand, an RFID reader device that reads out many devices at the same time, requires a higher throughput, but area and power consumption are less important. Active smart devices, such as contact smart cards do not face strict power constraints but timing and sometimes energy constraints. Main key figures of the PRESENT block cipher are area, throughput, and power consumption. We propose three implementations of PRESENT, so one can choose the architecture that meets the given requirements most suitable. The first architecture is round based as described in [3]. It is optimized in terms of area, speed, and energy. The second architecture uses pipelining technique and generates a high throughput. The third architecture is serialized and is minimized in terms of area and power consumption. In order to decrease the area requirements even further, all architectures can perform encryption only. This is sufficient for encryption and decryption of data when the block cipher is operated for example in counter mode. Besides this it allows a fairer comparison with other lightweight implementations. For example the landmark implementation of Feldhofer et al. [7]. Finally, using the round based architecture of PRESENT128, we present a cryptographic co-processor with encryption and decryption capabilities. Note that the choice of an appropriate I/O interface is highly application specific, while at the same time can have a significant influence on the area, power, and timing figures. In order to have a clearer estimation of the cryptographic core's efficiency we did therefore not implement any special input or output interfaces, but rather chose a natural width of 64-bit input, 64-bit output and 80 or 128-bit key input, respectively.

3.1 Round-Based Architecture

This architecture represents the direct implementation of the PRESENT top-level algorithm description in Figure 1, i.e. one round of PRESENT is performed in one clock cycle. The focus lies on a compact solution but at the same time with an eye on the time-area product. To save power and area a loop based approach is chosen. The balance between the 64-bit datapath and the used operations per clock cycle leads to a good time-area product. Due to the reuse of several building blocks and the round structure, the design has a high energy efficiency as well. The architecture uses only one substitution and permutation layer. So the datapath consists of one 64-bit XOR, 16 S-Boxes in parallel, and one P-Layer.

To store the internal state and the key, a 64-bit state register and an 80-bit key register are introduced. Furthermore an 80-bit 2-to-1 multiplexer and a 64-bit 2-to-1 multiplexer to switch between the load phase and the round computation phase are required. Key register, key input multiplexer, a 5-bit XOR, one S-Box and a 61-bit shifter are merged into the component key scheduling. It computes the round key on the fly. Figure 2 presents the signal structure of the round based approach for PRESENT. At first the key and the plaintext are stored into the accordant register. After each round the internal state is stored into the state register. After 31 rounds the state is finally processed via XOR with the last round key. The control logic is implemented as a Finite State Machine (FSM) and a 5-bit counter to count the rounds. The FSM also controls the multiplexers to switch between load and encryption phase.

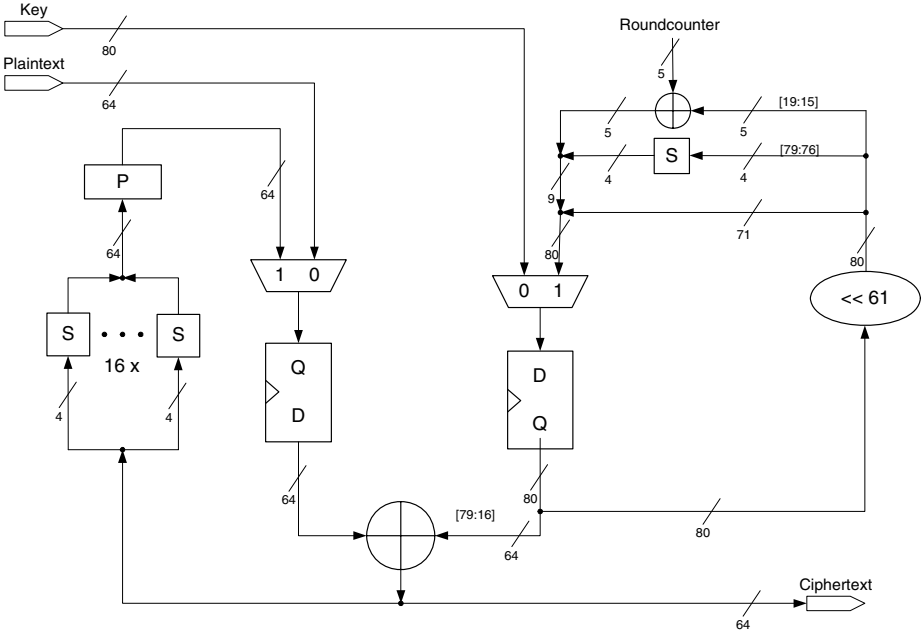


Fig. 2. Block diagram of the round-based PRESENT architecture

To reduce the used area and power we make use of clock gating. It can be applied to synchronous load enable registers, which are groups of flip-flops that are connected to the same clock and control signals. Normally a register is implemented by use of a flip-flop, a feedback loop, and a multiplexer. When this register bank maintains the same logic value through multiple clock cycles its clock network, the multiplexers and the flip-flops unnecessarily consume power. Clock gating eliminates the feedback nets and multiplexers inserting a latch and a 2-input gate in the clock net of the registers. The latch prevents glitches on the enable signal. By controlling the clock signal for the register bank, the need

for reloading the same value in the register through multiple clock cycles is eliminated. Clock gating reduces the clock network power dissipation, relaxes the data path timing, and reduces routing congestion by removing feedback multiplexer loops. For designs that have large multi-bit registers, clock gating can save power and further reduce the number of gates in the design. However, for smaller register banks, the overhead of adding logic to the clock tree might not compare favorably to the power saved by eliminating a few feedback nets and multiplexers.

3.2 Parallel Architecture

The main goal of the parallel design is to achieve a high throughput rate. Therefore the 31 time loop is unrolled, so all XORs, S-Boxes, and P-Layers are cascaded. This will lead to high area effort and power consumption, but also to high data throughput. The required round key is generated by taking the right bits from the 80-bit key and if necessary pass them through a S-Box or add a roundcounter value. All subkeys are available in parallel and no register is needed to hold the key. Figure 3 shows the signal diagram of the pipelined architecture. It consists of 32 XORs, 496 S-Boxes, and 31 P-Layeres for the datapath. The keypath consists of 31 S-Boxes and 31 XORs for key scheduling. The roundcounter input of the XOR is hard wired. First the given 64-bit plaintext and the first round key are xored. The result is split up into 16 4-bit blocks. Each block is processed by a 4-bit S-Box in parallel. The 64-bit P-Layer transposes the bits at the end of each the 31 rounds. Note that, the 32th round consists only of the XOR operation.

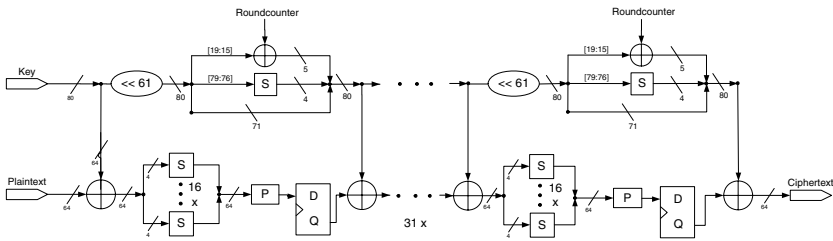


Fig. 3. Datapath of the pipelined parallel PRESENT architecture

This straight forward approach does not achieve a high maximum operating frequency. This results from the long critical path. The input signal has to propagate through all XOR and S-Box gates. The more gates belong to the path the higher is the resulting capacitance to be switched. So the time period for a switching event is stretched. To shorten the critical path, flip-flops as pipeline stages were installed after each P-Layer (see Figure 3). On the one hand this increases the chip area and power consumption, but on the other hand the maximum frequency can be raised significantly. We assume the key to be stable for many encryption operations. Thus roundkeys do not propagate through the pipeline and need not to be stored in additional FFs.

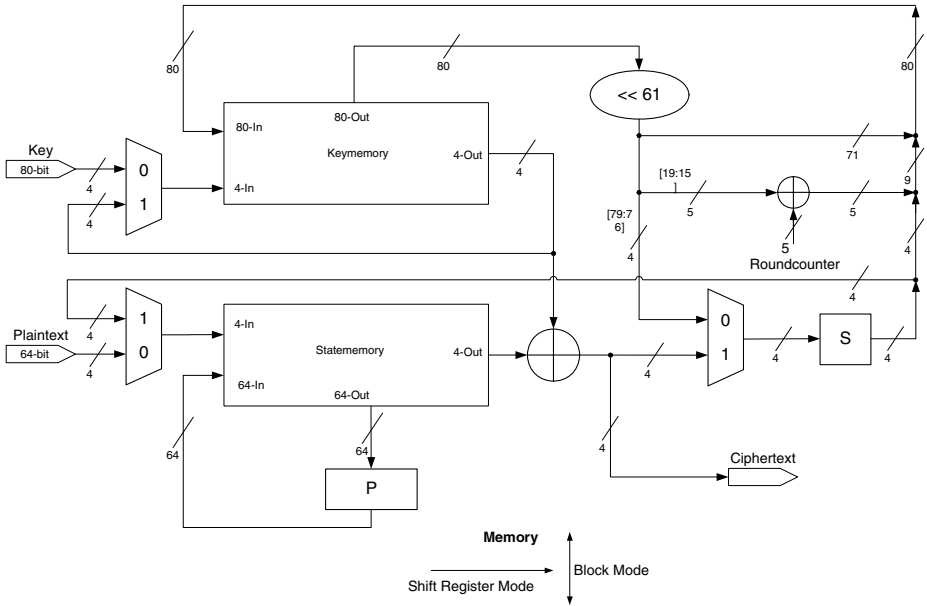


Fig. 4. Datapath of the serial PRESENT architecture

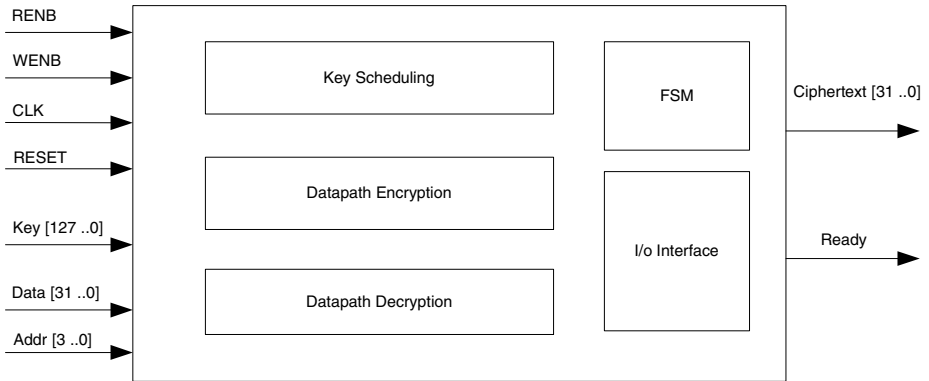


Fig. 5. Block diagram of PRESENT-128 coprocessor with 32-bit interface

3.3 Serialized Approach

This architecture is a further modification on the round based architecture described in Section 3.1. To save more chip area, the data structure is reduced to 4-bit. One of the most area consuming parts of PRESENT are the 16 S-Boxes in parallel. So only one of them is used to represent the substitution layer, which is also shared between the data path and the key scheduling. Another power and area consuming part are the large input multiplexers. We use a 4-bit interface

to read in key and plaintext. This area savings come at the disadvantage of a longer computation time. Only 4-bit are processed during one clock cycle and we need 20 clock cycles for initialization. An additional 4-bit counter upgrades the FSM to control the processing of the internal state. Therefore it takes additional 15 cycles to compute the substitution layer of each round. As one can see in Figure 4 the signal diagram still shows a 64-bit wide and a 80-bit wide path. The main problem is to serialize the permutation layer. So we choose a memory structure with two different operation modes. In the first mode it behaves like a shift register. During load phase and S-Box computation phase the 4-bit input is shifted to the left. The 4-bit output is appended at the beginning. If the P-Layer is computed all bits are read in parallel and the 64-bit wide or 80-bit wide input and output is used. Each memory element consists of scan flip-flops, i.e. a D-flip-flop with integrated multiplexer, which saves area compared to one normal D-flip-flop and a separated multiplexer. One further advantage is the reduced computation time, so we need only one clock cycle for the whole P-layer. A 4-bit computation scheme would lead to much more multiplexers. All together we need 17 clock cycles per round to compute the new state.

3.4 Crypto Coprocessor

To equip a smart device with cryptographic functions there are different ways to implement them. The first is to write software code. This solution requires RAM to store the program and inhibits the microcontroller while performing cryptographic algorithms. Another possibility is to implement the crypto part straight into the the microcontroller core. A more flexible way is to construct a cryptographic co-processor that is controlled by the main core. It uses a memory-like interface for communication. To get a compact and also fast solution we use the round based architecture with a modified finite state machine and added further multiplexers. Now the plaintext is loaded in 32-bit blocks. As far as we know this is the maximum bit width of microcontrollers for smart devices. The co-processor is controlled by write and read enable signals. The address signal selects the different bit blocks and encryption or decryption mode. Figure 5 illustrates the interfaces and the units.

4 Evaluation of the Results

In this section we first describe the used design flow and the metrics. Subsequently we compare our implementation results for the three scenarios low cost passive smart devices, low cost active smart devices, and high end smart devices. We considered the following optimization goals for the three scenarios: low cost and passive smart devices should be optimized for area and power constraints and low cost and active smart devices for area, energy, and time constraints. Note that in our methodology high end devices are always contact smart cards and hence should be optimized for time and energy constraints. Therefore we do not distinguish between passive and active high end smart devices.

4.1 Metrics and Used Design Flow

All architectures were developed and synthesized by using a script based design flow. We used MentorGraphics FPGA Advantage 8.1 for HDL source code construction and functional verification. Then the RTL description was synthesized with Synopsys *Design Compiler* Z-2007.03-SP5, which was also used to generate the area, timing, and power estimation reports. The main effort of synthesis process was area optimization. The S-Box is described as boolean equation which leads to a combinatorial logic implementation. The P-Layer is only simple wiring, which is not very costly in hardware. We used three different standard cell libraries with different technology parameters: a 350 nm technology MTC45000 from AMIS, a 250 nm technology SESAME-LP2 from IHP, and a 180 nm technology UMCL18G212D3 from UMC. Each of them consists of a different amount of cells and not all logical functions are implemented. This fact will lead to different area result expressed in GE. Following definitions of metrics were used:

Area: This metric represents the amount of area normalized to the area of one NAND gate. This ratio is expressed in GE.

Cycles: Number of clock cycles to compute and read out the ciphertext.

Throughput: The rate at which new output is produced with respect to time. The number of ciphertext bits is divided by the needed cycles and multiplied by the operating frequency. It is expressed in bits-per-second. With increasing frequency the throughput will increase, too.

Power: The power consumption is estimated on the gate level by PowerCompiler¹. It consists of two major components: the static power which is proportional to the area and the fabrication process. The dynamic power is proportional to the switching activity (switching event probability and operating frequency). Both components also depend on the supply voltage.

Current: The power consumption divided by the typical core voltage of the process. These are for AMI 3.3V, for IHP 2.5V, and for UMC 1.8V.

Throughput to area ratio: This representation is used as a measure of design efficiency.

Maximum frequency: There are many connections between the input and output pins. The delay of each gate forms a timing path for the signals. The slowest path will set the upper bound of clock frequency. Note that it might be possible to increase the max. frequency, but this will also increase area and power.

The interested reader can find more detailed tables with syntheses results in the appendix.

4.2 Low Cost Passive Smart Devices

Table 4.1 shows the synthesis results for 100 kHz clock frequency, which is a typical operating frequency of RFID tags. Smart devices with integrated

¹ Note that power estimations on the transistor level are more accurate. However, this also requires further design steps in the design flow, e.g. place&route.

contactless functionality have strict area and power constraints. For this purpose we propose a serialized implementation which will consume low area and power resources. Our serial implementation uses about 1000 GE of area. To the best of our knowledge this is the smallest implementation of a cryptographic algorithm with a moderate security level. Even implementations of the stream ciphers Grain80 and Trivium require more area (1294 GE and 1857 GE, respectively [9]). For comparison with block ciphers we choose two AES implementations with a reduced datapath from Feldhofer et al. [6] and Hämäläinen et al. [10]. Furthermore there exists only a reduced datapath implementation of the lightweight block cipher SEA without key scheduling component and control logic. Note that a similar implementation with PRESENT would only require around 40 GE in 0.18 μ m UMC technology. The power consumption of our implementations show a large variation depending on the core voltage of the library, but the 0.18 μ m technology consumption is still the lowest compared to the other architectures. Note that power figures are highly technology dependent, therefore a fair comparison is only possible if the same technology was used.

Table 1. Implementation results of minimal datapath architectures

Cipher	Tech. [μ m]	Datapath [Bit]	Freq. [MHz]	Area [GE]	Throughp. [Kbps]	Cycles	Power [μ W]
PRESENT-80	0.35	4	0.1	1,000	11.4	563	11.20
PRESENT-80	0.25	4	0.1	1,169	11.4	563	4.24
PRESENT-80	0.18	4	0.1	1,075	11.4	563	2.52
Feldhofer AES [6]	0.35	8	0.1	3400	12.4	1032	4.50
Hämäläinen AES [10]	0.13	8	80	3100	121	160	-
SEA [15]	0.13	8	0.1	449		50	3.22
better is				lower	higher	lower	lower

4.3 Low Cost Active Smart Devices

The second scenario targets standard smart cards. To reduce fabrication costs these cards are also area constraint. But in comparison to the prior scenario the crypto core draws his energy from a battery of a pervasive device or via the physical contact of the reading device. So the execution time is of major interest. The round based implementation shows a good trade off between area, time, throughput, and energy consumption. It does not consume significant more area and energy than the serial one, but needs much less clock cycles for computation. The results are compared to other known round based implementations that means a new internal state is computed every clock cycle. There are results for the ICEBERG [21] and the HIGHT [11] block cipher. Both of them use a 64-bit datapath architecture. In Mace et al. [15] different ASIC implementations of SEA had been characterized. We choose the 96-bit architecture for better comparison to the other datapaths. The results in Table 2 illustrate the very compact design of the PRESENT block cipher. Even the -normalized to 10 MHz.- throughput is only

Table 2. Implementation results of the round based datapath architectures

Cipher	Tech. [μm]	Datapath [Bit]	Freq. [MHz]	Area [GE]	Tput [Mbps]	Energy/Bit [pJ/bit]	Power [μW]
PRESENT-80	0.35	64	10	1561	20.6	170.5	3520.0
PRESENT-80	0.25	64	10	1594	20.6	21.1	436.0
PRESENT-80	0.18	64	10	1705	20.6	3.7	77.1
SEA [15]	0.13	96	250	3758	258.0	19.8	5102.0
ICEBERG [15]	0.13	64	250	7732	1000.0	9.6	9577.0
HIGHT [11]	0.25	64	80	3048	150.6	-	-
better is				lower	higher	lower	lower

Table 3. Implementation results of pipelined architecture @ 10 MHz

Library	Area [GE]	Power [μW]	Tput/Area [kpbs/ μm^2]	crit. Path [ns]	max Freq. [GHz]	max .Tput [Mbps]
AMI 0.35 μm	24,345.87	81295.00	0.486811614	12.80	0.1	5,000.0
IHP 0.25 μm	25,193.00	11659.00	0.900080911	4.78	0.2	13,389.1
UMC 0.18 μm	27,027.69	6888.00	2.446979668	6.26	0.2	10,223.6
better is	lower	lower	higher	lower	higher	higher

Table 4. Implementation results of co-processor architectures

Cipher	Tech. [μm]	Datapath [Bit]	max Freq. [MHz]	Area [GE]	Throughp. [Mbps]	Cycles
PRESENT-128	0.35	32	143	2,681	234	39
PRESENT-128	0.25	32	141	2,917	231	39
PRESENT-128	0.18	32	323	2,989	529	39
PRESENT-128	0.35	8	131	2,587	133	63
PRESENT-128	0.25	8	121	2,851	123	63
PRESENT-128	0.18	8	353	2,900	359	63
CAST AES [4]	0.18	32	300	124,000	872	44
Satoh AES [20]	0.11	32	131	54,000	311	54
Pramstaller AES [19]	0.6	32	50	85,000	70	92
better is			higher	lower	higher	lower

outperformed by the ICEBERG implementation., but again, we do not consider high throughput as highly relevant for this device class.

4.4 High End Active Smart Devices

In the third scenario there are no limitations for energy consumption. The task of the co-processor is to relieve the micro controller of the cryptographic computations. The design of this assistant should deliver results fast and consume as less area as possible to be cost-effective. One approach is to use a pipelined

architecture. But Table 3 discloses that the pipelined implementation generates a very high throughput at the expense of area and power. The basic message is that scaling of operation frequency has a great impact on power consumption. The area is barely affected by this circumstance, because we chose an area optimize synthesis approach. If we get to higher frequencies the capacitances will become increasingly important. So cells with a higher driving strength must be used to drive the load and the area will increase conspicuously. In addition one has to be aware of the input/output interface. Up to now there exist only smart cards with 32-bit micro controllers. The best choice is to implement a round based architecture with an 32-bit I/O interface. In literature can be found several AES implementations that are up to the mark. We compare the PRESENT implementations to Pramstaller et al. [19] and Satoh et al. [20]. Also a commercial solution by Cast Inc. [4] is listed. Table 4 shows the results for the different implementations. As there are many smart cards equipped with 8-bit microcontrollers we list the results for an 8-bit interface, too. The PRESENT co-processor is much more compact than the other implementations and also needs less clock cycles to compute the ciphertext.

5 Conclusions

In this paper we have pointed out that there is, due to harsh cost constraints inherent of mass deployment, a strong need for area optimized implementation of cryptographic algorithms. Furthermore, we presented the implementation results of three different architectures of the block cipher PRESENT. The pipelined version achieves a high throughput to area ratio but also consumes the most area and current compared to the other architectures. Therefore this architecture may be used in high end smart devices and the back end systems. The serial version can be implemented with as few as 1000 GE, which is to the best of our knowledge the smallest implementation of a cryptographic algorithm with a moderate security level. However, this significant area savings come at the disadvantage of a long processing time of 563 cycles. This architecture is best suited for low cost passive smart devices such as passive RFID tags and contactless smart cards.

Interestingly, the round version draws for two of the three different libraries nearly the same current consumption. It requires about 50% more area but also achieves a relatively high throughput rate compared to the serialized architecture. This in turns yields a good energy consumption per encryption, hence this architecture is well suited for low cost active smart devices such as wireless sensor nodes, RFID reader handhelds, and contact smart cards. Furthermore this architecture can be used to construct a cryptographic coprocessor with very low area consumption and a high throughput.

References

1. Keeloq algorithm (November 2006), <http://en.wikipedia.org/wiki/KeeLoq>
2. Bogdanov, A.: Attacks on the KeeLoq Block Cipher and Authentication Systems. In: 3rd Conference on RFID Security 2007 (RFIDSec 2007) (2007)

3. Bogdanov, A., Leander, G., Knudsen, L.R., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT - An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727 Springer, Heidelberg (2007)
4. Cast Inc. Cast aes32-c, <http://www.cast-inc.com>
5. Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Hong, D., Sung, J., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)
6. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. In: Information Security, IEE Proceedings, vol. 152, pp. 13–20 (October 2005)
7. Feldhofer, M., Dominikus, S., Wolkerstorfer, J.: Strong Authentication for RFID Systems Using the AES Algorithm (2004)
8. Finkenzeller, K.: RFID Handbook - Fundamentals and Applications in Contactless Smart Cards and Identification, 2nd edn. John Wiley and Sons. Ltd., Chichester (2003)
9. Good, T., Benaïssa, M.: Hardware Results for selected Stream Cipher Candidates. In: State of the Art of Stream Ciphers 2007 (SASC 2007), Workshop Record (February 2007)
10. Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D.: Design and implementation of low-area and low-power aes encryption hardware core. In: DSD, pp. 577–583 (2006)
11. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device (2006)
12. Kerckhoff, A.: La cryptographie militaire. *Journal des sciences militaires* IX, 5–38 (1883)
13. Leander, G., Paar, C., Poschmann, A., Schramm, K.: New Lightweight DES Variants. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 196–210. Springer, Heidelberg (2007)
14. Lim, C., Korkishko, T.: mcrypton - a lightweight block cipher for security of low-cost rfid tags and sensors. In: Kwon, T., Song, J., Yung, M. (eds.) WISA 2005. LNCS, vol. 3786, pp. 243–258. Springer, Heidelberg (2006)
15. Mace, F., Standaert, F.-X., Quisquater, J.-J.: ASIC Implementations of the Block Cipher SEA for Constrained Applications. In: Proceedings of the Third International Conference on RFID Security - RFIDSec 2007, Malaga, Spain, pp. 103–114 (2007)
16. N.A. Contactless Specifications for Payment Systems - EMV Contactless Communication Protocol Specification. Version 2.0, EMV (August 2007), <http://www.emvco.com/specifications.asp>
17. N.A. Contactless Specifications for Payment Systems - Entry Point Specification. Draft 1.0, EMV (October 2007), <http://www.emvco.com/specifications.asp>
18. Nohl, K., Ploetz, H.: Mifare - little security, despite obscurity. Talk at the 24th Chaos Communication Congress (December 2007)
19. Pramstaller, N., Mangard, S., Dominikus, S., Wolkerstorfer, J.: Efficient aes implementations on asics and fpgas. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 98–112. Springer, Heidelberg (2005)

20. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact rijndael hardware architecture with s-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
21. Standaert, F.-X., Piret, G., Gershenfeld, N., Quisquater, J.-J.: Sea: A scalable encryption algorithm for small embedded applications. In: Domingo-Ferrer, J., Posegga, J., Schreckling, D. (eds.) CARDIS 2006. LNCS, vol. 3928, pp. 222–236. Springer, Heidelberg (2006)
22. Visa. Visa payWave FAQ (accessed on 15.02.2008), www.visa.com

Appendix

Following abbreviations are used in the subsequent tables

Cur - Current

Tput/Area - Throughput/Area

mFreq - maximum Frequency

mTput - maximum Throughput

Table 5. Implementation results of round @ 100 kHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur [μA]	Tput/Area [$\text{kbps}/\mu\text{m}^2$]	Path [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	1,524.77	82,338	33.40	10.12	0.0024	1.53	0.65	1,307.2
IHP 0.25 μm	1,594.25	44,996	4.84	1.94	0.0044	0.72	1.39	2,777.8
UMC 0.18 μm	1,650.30	15,970	3.86	2.14	0.0125	4.57	0.22	437.6
better is	lower	lower	lower	lower	higher	lower	higher	higher

Table 6. Implementation results of round @ 10 MHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur [μA]	Tput/Area [$\text{kbps}/\mu\text{m}^2$]	Path [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	1,560.5	84,268	3520.0	1066.7	0.2450	1.23	0.81	1,678.5
IHP 0.25 μm	1,594.2	44,996	436.0	174.4	0.4588	0.61	1.64	3,384.5
UMC 0.18 μm	1,706.0	16,509	77.1	42.8	1.2506	0.51	1.96	4,048.1
better is	lower	lower	lower	lower	higher	lower	higher	higher

Table 7. Implementation results of pipeline @ 100 kHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur [μA]	Tput/Area [$\text{kbps}/\mu\text{m}^2$]	Path [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	24,247	1,309,354	772.0	233.9	0.0049	13.84	0.07	4,624.3
IHP 0.25 μm	25,193	711,047	121.0	48.4	0.0090	4.98	0.20	12,851.4
UMC 0.18 μm	27,009	261,366	72.2	40.1	0.0245	6.78	0.15	9,439.5
better is	lower	lower	lower	lower	higher	lower	higher	higher

Table 8. Implementation results of pipeline @ 10 MHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur. [μA]	Tput/Area [kpbs/ μm^2]	Path [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	24,346	1,314,677	81295.0	24634.8	0.4868	12.8	0.08	5,000
IHP 0.25 μm	25,193	711,047	11659.0	4663.6	0.9001	4.78	0.21	13,389
UMC 0.18 μm	27,028	261,547	6888.0	3826.7	2.4470	6.26	0.16	10,224
better is	lower	lower	lower	lower	higher	lower	higher	higher

Table 9. Implementation results of serial @ 100 kHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur. [μA]	Tput/Area [kpbs/ μm^2]	Path [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	999.5	53,974	11.20	3.39	0.0002	1.89	0.5	60.1
IHP 0.25 μm	1,168.8	32,987	4.24	1.70	0.0003	0.66	1.5	172.2
UMC 0.18 μm	1,075.0	10,403	2.52	1.40	0.0011	0.9	1.1	126.3
better is	lower	lower	lower	lower	higher	lower	higher	higher

Table 10. Implementation results of serial @ 10 MHz

Library	Area [GE]	Area [μm^2]	Power [μW]	Cur. [μA]	Tput/Area [kpbs/ μm^2]	cPath [ns]	mFreq [GHz]	mTput [Mbps]
AMI 0.35 μm	1,001.19	54,064	1123.00	340.30	0.0210	1.44	0.69	78.9
IHP 0.25 μm	1,168.75	32,987	421.00	168.40	0.0345	0.62	1.61	183.3
UMC 0.18 μm	1,074.98	10,403	247.00	137.22	0.1093	0.8	1.25	142.1
better is	lower	lower	lower	lower	higher	lower	higher	higher

Fast Hash-Based Signatures on Constrained Devices

Sebastian Rohde¹, Thomas Eisenbarth¹, Erik Dahmen², Johannes Buchmann²,
and Christof Paar¹

¹ Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany

{rohde,eisenbarth,cpaar}@crypto.rub.de

² Technische Universität Darmstadt

Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{dahmen,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. Digital signatures are one of the most important applications of microprocessor smart cards. The most widely used algorithms for digital signatures, RSA and ECDSA, depend on finite field engines. On 8-bit microprocessors these engines either require costly coprocessors, or the implementations become very large and very slow. Hence the need for better methods is highly visible. One alternative to RSA and ECDSA is the Merkle signature scheme which provides digital signatures using hash functions only, without relying on any number theoretic assumptions. In this paper, we present an implementation of the Merkle signature scheme on an 8-bit smart card microprocessor. Our results show that the Merkle signature scheme provides comparable timings compared to state of the art implementations of RSA and ECDSA, while maintaining a smaller code size.

Keywords: Embedded security, hash based cryptography, Merkle signature scheme, digital signatures.

1 Motivation

Smart cards are used in many areas of every day life. Application areas include payment systems, electronic health cards and SIM cards for mobile phones. With the advent of contactless smart cards, new and important fields of application have recently emerged, like the electronic passport, which is now deployed in many countries, especially in Europe and the US. Other countries, like Belgium, also issue electronic ID cards to their citizens [1].

The most important application of smart cards is secure identification and authentication. Many of the above mentioned applications have a need for strong security. All these requirements are met by digital signatures. Digital signatures provide authenticity, integrity and support for non-repudiation of data and are often used in identification and authentication protocols for smart cards.

Since smart cards are usually provided in high quantities, there is also a need to keep costs as low as possible. This is one of the reasons why most of the microprocessor cards in use are still equipped with small and cheap 8-bit CPUs. These small 8-bit microprocessors are constrained in program memory (flash or ROM), RAM, clock speed, register width, and arithmetic capabilities.

Common signature schemes such as RSA and ECDSA require operations in a finite field for the signature generation and verification. For efficient implementations in smart cards, costly coprocessors that implement the field arithmetic are required. In 1979 Merkle proposed a signature scheme that requires only hash function evaluations for the signature generation and verification [20]. Since software implementations of hash functions are much more efficient than software implementations of finite field arithmetic, the Merkle signature scheme (MSS) is a good candidate for implementations on small microprocessors without cryptographic coprocessors. Another benefit of the MSS is the fact that its security relies only on the cryptographic properties of the used hash function and not on additional number theoretic assumptions. If the hash function used for the MSS is found insecure, it can be replaced by a secure one to obtain a new and secure instance of the MSS.

Our Contribution. In this paper we present an implementation of the Merkle signature scheme for 8-bit Atmel AVR microcontrollers, e.g. smart card processors from the AT90SCxxx family. Our implementation is highly scalable and can be configured to provide an ideal tradeoff between security, execution times, and memory requirements for the specific use case. We will show that our implementation of the MSS performs excellently when compared to RSA and ECDSA. Our implementation has a smaller code size and faster verification times. The signature generation is faster than RSA and comparable to ECDSA. Further performance improvements are reached by utilizing a symmetric crypto engine such as an AES hardware acceleration.

For the underlying hash functions we use constructions that are based on the AES block cipher. Such hash functions have two advantages compared to dedicated hash functions: (1) they have a small block size which is more suitable for the MSS and (2) they are more efficient in size and speed.

Related Work. Gura *et al.* showed the feasibility of public key cryptography on constrained 8-bit microcontrollers. Their implementation of RSA-1024 and RSA-2048 showed that digital signatures are feasible on 8-bit platforms even without expensive crypto-coprocessors. Further research regarding digital signatures on constrained 8-bit devices has been performed in the field of wireless sensor networks. Liu and Ning published a full ECC engine called TinyECC which also does not require a coprocessor. They implemented the 160-bit elliptic curve *secp160r1*. Winternitz one-time signatures have also been proposed to be used in wireless sensor networks for signing short messages (≤ 80 bit) [18]. The proposed solution, however, uses a public key management that is not

applicable to smart cards. Others [24,7] show possible use cases for MSS on constrained devices without making any suggestions regarding the implementation.

Organization. The paper is organized as follows: Section 2 gives an overview of the MSS and the used hash functions. Section 3 explains the target platform and details about the implementation. Section 4 presents performance results and a comparison. Section 5 elaborates a possible performance gain when using an AES hardware acceleration. Section 6 states our conclusion.

2 Preliminaries

In this section we describe the details of the variant of the Merkle signature scheme [20] we use for our implementation. In summary we use the Winternitz one-time signature scheme (W-OTS) [9] to sign the data, the ideas for efficient one-time signature key generation of [4] and the algorithm from [6] for the computation of the authentication paths. We use two different hash functions based on the AES block cipher, both with 128-bit block length. We use a 256-bit hash function for the initial hashing (digest creation) of the data to be signed and a 128-bit hash function for the one-time signature scheme and the Merkle tree. Details on the construction of these hash functions are described in Section 2.2.

2.1 The Merkle Signature Scheme

We now describe the three algorithms for the key generation, signature generation, and verification. In the following, let $F : \{0,1\}^* \rightarrow \{0,1\}^{128}$ and $G : \{0,1\}^* \rightarrow \{0,1\}^{256}$ be cryptographic hash functions.

Key Generation. The first step of the key generation is to decide how many signatures should be generated with this key pair. We choose the parameter $H \geq 2$ to be able to generate 2^H signatures. The next step is to generate 2^H W-OTS key pairs. For the W-OTS key generation, we apply the approach of [4] and use the following forward secure pseudo random number generator.

$$\text{PRNG} : \{0,1\}^{128} \rightarrow \{0,1\}^{128} \times \{0,1\}^{128}, \text{SEED}_{\text{in}} \mapsto (\text{SEED}_{\text{out}}, \text{RAND}).$$

As suggested in [5], we use the hash based PRNG proposed in [12], i.e.

$$\text{RAND} \leftarrow F(\text{SEED}_{\text{in}}), \text{SEED}_{\text{out}} \leftarrow (1 + \text{SEED}_{\text{in}} + \text{RAND}) \bmod 2^{128}.$$

The MSS private key is an 128-bit seed SEED chosen uniform at random. This seed is fed to the PRNG to compute the initial seed SEED_{W-OTS} that we use to generate first W-OTS signature key:

$$(\text{SEED}, \text{SEED}_{\text{W-OTS}}) = \text{PRNG}(\text{SEED}). \quad (1)$$

Doing so, SEED is updated and can be used to compute the initial seeds for upcoming W-OTS signature keys. Depending on the Winternitz parameter w , the W-OTS signature key consists of $t = t_1 + t_2$ 128-bit strings, where

$$t_1 = \left\lceil \frac{256}{w} \right\rceil, \quad t_2 = \left\lceil \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rceil.$$

The W-OTS signature key is the sequence $X = (x_1, \dots, x_t)$, that consists of t bit strings each of length 128-bit. It is computed using the PRNG as

$$(\text{SEED}_{\text{W-OTS}}, x_i) = \text{PRNG}(\text{SEED}_{\text{W-OTS}}) \quad (2)$$

for $i = 1, \dots, t$. The W-OTS verification key is $Y = F(y_1 \parallel \dots \parallel y_t)$, where $y_i = F^{2^w - 1}(x_i)$, i.e. the hash function F is applied $2^w - 1$ times to x_i for $i = 1, \dots, t$.

The 2^H W-OTS verification keys are the leaves of the Merkle tree. The inner nodes are computed using the following construction rule: a parent node is the hash of the concatenation of its left and right children, i.e.

$$\text{Node}_{\text{parent}} = F(\text{Node}_{\text{left child}} \parallel \text{Node}_{\text{right child}}).$$

By applying this rule iteratively the root of the Merkle tree, which is also the MSS public key, is obtained.

Signature Generation. To sign some data, the first step is to compute its 256-bit digest: $d = G(\text{data})$. The W-OTS signature keys are used sequentially. We describe the generation of the s th signature, $s \in \{0, \dots, 2^H - 1\}$. The s th W-OTS signature key is computed from the seed SEED as described in Equations (1) and (2). We always update the seed in the private key and therefore one invocation of the PRNG suffices to obtain the initial seed $\text{SEED}_{\text{W-OTS}}$ to compute the s th W-OTS signature key. The Winternitz signature of d is then computed as follows: (1) split the binary representation d into t_1 blocks b_1, \dots, b_{t_1} each of length w . (2) Consider b_i as the integer encoded by this block in binary and compute $c = \sum_{i=1}^{t_1} (2^w - b_i)$. (3) Split the binary representation c into t_2 blocks b_{t_1+1}, \dots, b_t each of length w . If the bit-length of c or d is no multiple of w we pad with zeros to the left. The Winternitz signature of d is then given as $\sigma_{\text{W-OTS}}(d) = (\sigma_1, \dots, \sigma_t)$, where $\sigma_i = F^{b_i}(x_i)$, for $i = 1, \dots, t$. The s th MSS signature of d is given as

$$\sigma_s(d) = (s, \sigma_{\text{W-OTS}}(d), (a_0, \dots, a_{H-1})).$$

The sequence (a_0, \dots, a_{H-1}) is the authentication path for the s th leaf, i.e. the s th W-OTS verification key. It is defined as the siblings of all nodes on the path from the s th leaf to the root of the Merkle tree, see Figure 1. For the computation of authentication paths we use the BDS algorithm from [6]. This algorithm is constructed such that the authentication path for the currently used leaf is already available and the upcoming authentication paths are prepared after the MSS signature is computed. The BDS algorithm uses a parameter $K \geq 2$ which decides how many nodes close to the root are stored during the initialization to reduce the computational cost. The initialization of this algorithm, that requires certain tree nodes to be stored, is done during the MSS key generation.

Signature Verification. The first step of the signature verification is again to compute the digest of the data that was signed: $d = G(\text{data})$. Then d and its

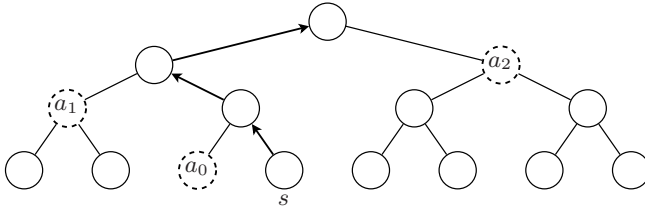


Fig. 1. Example of the Merkle signature scheme for $H = 3, s = 3$. Dashed nodes denote the authentication path for the s th leaf. The arrows indicate the path from the s th leaf to the root.

Winternitz signature are used to compute the s th leaf as follows: Repeat steps (1)-(3) of the Winternitz signature generation to obtain b_1, \dots, b_t . The s th leaf φ is computed as

$$\varphi = F(F^{2^w-1-b_1}(\sigma_1) \parallel \dots \parallel F^{2^w-1-b_t}(\sigma_t))$$

Then the path from the s th leaf to the root and the root itself is recomputed using the authentication path and the index s :

$$\varphi = \begin{cases} F(\varphi \parallel a_h), & \text{if } s/2^h \equiv 0 \pmod 2 \\ F(a_h \parallel \varphi), & \text{if } s/2^h \equiv 1 \pmod 2 \end{cases}$$

for $h = 0, \dots, H - 1$. If the computed root matches the signers public key, the signature is valid.

Time and Memory Requirements. We now estimate the number of evaluations of F required for the key generation, signature generation and verification. We also estimate the storage requirements of the public and private key and the signatures.

The MSS key generation requires the computation of 2^H leaves or W-OTS key pairs and $2^H - 1$ evaluations of F to compute the root. The computation of one leaf costs $t(2^w - 1) + 1$ evaluations of F and $t + 1$ calls to the PRNG. Using that one call to the PRNG costs as much as one evaluation of F , the key generation in total requires $2^H(t2^w + 3) - 1$ evaluations of F . The public key requires 128 bits of memory.

For each signature, the BDS algorithm requires at most $(H - K)/2 + 1$ leaves, $3(H - K - 1)/2 + 1$ evaluations of F and $H - K$ calls to the PRNG to compute upcoming authentication paths. If s is even the BDS algorithm requires $(H - K)/2 + 1$ leaves to be computed. One of these leaves is the s th leaf. Since the Winternitz signature of the data just signed using the s th W-OTS key is an intermediate value during the computation of the s th leaf, the generation of this Winternitz signature needs no additional calculations in this case. If s is odd, the BDS algorithm requires only $(H - K)/2$ leaves to be computed and the Winternitz signature of the data must be computed separately. Since the generation of a Winternitz signature requires less computations than a leaf,

the above cost for the BDS algorithm also represent the total cost for signing. Hence, the total cost for signing in terms of evaluations of F is at most $t2^w(H - K - 2)/2 + (7H - 7K + 3)/2$. The BDS algorithm needs to store $3.5H - 3K + 2^K - 2$ nodes of the Merkle tree and $2(H - K)$ seeds which we store as part of the private key. Together with the 128-bit seed used to generate the signature keys, the size of the private key is given as $(5.5H - 5K + 2^K - 1) \cdot 128$ bits. The size of the signature is given as $(t + H) \cdot 128$ bits. $t \cdot 128$ bits for the Winternitz signature and $H \cdot 128$ bits for the authentication path.

The signature verification on average requires $t(2^w - 1)/2$ evaluations of F to compute the s th leaf and H evaluations of F to recompute the path to the root and the root itself. The signature generation and verification also require one evaluation of G to compute the initial digest d of the data.

The above formulas show that the Winternitz parameter w provides a time-memory trade-off of the signature size and the key and signature generation times. However, the key and signature generation times of the W-OTS keys are exponential in w , while the signature size decreases only linearly in w . Therefore w should not be chosen too large. Also the output length of the hash functions F and G must be chosen carefully since the size of a Winternitz signature linearly depends on their product. Our choice of 128 and 256 bit yields moderate signature sizes and, as we will explain in the following, high practical security.

Security. The MSS is provably secure against adaptive chosen message attacks, if the used hash function is collision resistant [8]. However, to forge a MSS signature in practice the attacker is required to compute preimages and second-preimages. Therefore the practical security of the MSS currently relies on the preimage and second-preimage resistance of the used hash function [21]. From a practical point of view, the 128-bit hash function F we use for the W-OTS and the Merkle tree provides 128-bit security. Collision resistance is definitely required for the initial hashing of the data to sign. This is why we use the 256-bit hash function G , which provides 128-bit security against collision attacks that exploit the birthday paradox.

2.2 Hash Functions

In this section we present the hash functions that are used in our scheme. Furthermore we show that single and double block length constructions are the better choice when used in conjunction with digital signatures. Relatively short input block lengths and the resulting speed make them better suited for implementations on constrained devices. Public key and private key sizes are proportionally dependent on the hash length of F . As stated earlier, a short value of 128 bit offers adequate security for this scenario while staying within reasonable memory limits. For our scheme, we used the AES algorithm which is specified with a block length of 128 bit. Using AES in a double block length construction leads to a hash length of 256 bit.

Using block ciphers as hash functions in digital signature schemes is also appealing because one primitive can be used for three applications: encryption,

generation of hashes, and digital signatures. In addition block ciphers are much better known and analyzed than dedicated hash functions.

Single Block Length Construction. The single block length hash in our scheme is constructed using a Matyas-Meyer-Oseas (MMO) construction [19]. The MMO construction is recursively defined as $f_{i+1} = E_{f_i}(M_i) \oplus M_i$ with E being the encryption function, M_i as the current message block and f_0 being an initialization vector (See Figure 2). In a hash signature scheme this variant for constructing the hash benefits from the fact that the encryption function always uses the same key (an initialization vector) for the first block.

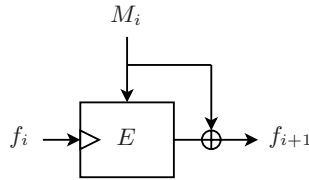


Fig. 2. Single block length compression function due to [19]. The output of the block cipher E is xored with the message block M_i . f_i, f_{i+1} , and M_i are each of bit length 128.

Double Block Length Construction. For applications such as the initial digest generation in a signature scheme, collision resistance is needed and the security of single block length (SBL) constructions is not sufficient. For our implementation, we use MDC-2 double length construction specified in the ISO/IEC 10118-2 standard. The standard envisions the usage of DES, but there is a variant using AES-128 [23], as depicted in Figure 3. This construction takes a block cipher with block length n bit and produces a hash function with $2n$ bit output length. In [22] the authors show, that an adversary needs at least $2^{3n/5}$ oracle queries to find a collision. However, the best practical attacks require 2^n queries.

The double block length construction is only used for initial digest generation. It can easily be replaced by a dedicated hash function resulting in a negligible performance loss but an increased code size.

Comparison to Dedicated Hash Functions. SBL and DBL constructions are much better suited for hash-based signature schemes than dedicated hash functions. A hash function with 512 bit (e.g. whirlpool) length would yield a highly inefficient signature scheme. At the same time it is also interesting to note that dedicated hash functions are optimized for large amounts of data as can be seen by the comparatively large block size (512 bit). With the MSS, the input for the hash function has mainly the same size as the output value. This is one of the reasons why dedicated hash functions provide suboptimal performance for appliances in hash based signature schemes.

In addition large block sizes reduce the speed of implementations on the AVR microcontroller platform since the state cannot be held completely in the registers of the processor. In Table 1 we provide a comparison of various hash

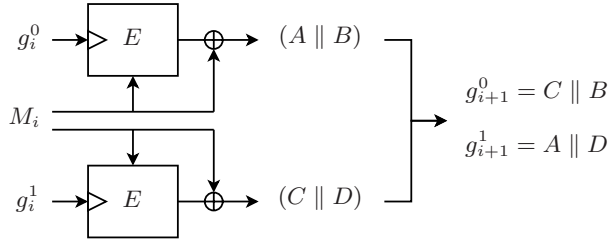


Fig. 3. Double block length compression function due to [23]. The outputs of the block cipher E are xored with the message block M_i and permuted. $g_i^0, g_i^1, g_{i+1}^0, g_{i+1}^1$, and M_i are each of bit length 128.

functions and their performance. The results for the dedicated hash functions are taken from [13] and [15].

Concerning the cycles required to hash one block, the block cipher based hash functions provide much better performance. This is due to the large block size of 512 bit used by dedicated hash functions to allow efficient hashing of large amounts of data. This is clarified by the column “cycles per byte” which shows that dedicated hash functions and block cipher based hash functions require a similar time to hash one input byte. However, for the Merkle signature scheme and our choice of parameters most of the time only blocks of length up to 256 bits must be hashed, which requires about 8,000 cycles when using the SBL construction. Hence, SBL constructions are a better choice than dedicated hash functions for the Merkle signature scheme on the target platform.

Table 1. Performance of hash function implementations on the AVR platform

Hash function	bit length per		msec per	cycles per	
	output	block		block	byte
SHA1 [13]	160	512	3.9	63,000	984
SHA1 [15]	160	512	2.6	41,113	642
SHA256 [15]	256	512	3.4	54,196	847
MD5 [13]	128	512	1.5	23,568	368
AES-SBL	128	128	0.3	4,081	255
AES-DBL	256	128	0.5	8,104	507

3 Implementation Details and Target Platform

Target Platform. Our implementation is designed for 8-bit AVR microcontrollers, a popular family of 8-bit RISC microcontrollers. The Atmel AVR processors offer clock speeds of up to 16MHz, a few KBytes of SRAM, up to tens of KBytes of EEPROM and additional flash or mask ROM for program memory. Besides the AVR smart card processors AT90SCxxx [2], AVRs are also available

as general purpose microcontrollers with a wide use in many embedded applications. One example is the Atmel ATmega128 microcontroller [3] often used for wireless sensor networks.

The devices of the AVR family have 32 general purpose registers of 8-bit word size. Most of the 130 instructions of the microcontroller are one-cycle. AVR microcontrollers can be programmed in AVR-assembler and in C.

The implementation of this project is designed to be executable on any AVR processor providing 4 KBytes SRAM, about 4 KBytes EEPROM and at least 8 KBytes of program memory. However, for platforms that are even more constrained in available SRAM, our scheme can also be altered to operate on systems with less SRAM. For our implementation the AVR was clocked within specification limits at 16 MHz. We chose to use assembler for performance critical routines such as some cryptographic primitives and C to glue these routines together.

AES Implementation. An efficient AES implementation for the AVR platform is available at [1]. It is licensed under the GPL. We modified this implementation to make it even smaller and faster. In this section we describe our modifications and improvements concerning this AES algorithm.

The RijndaelFurious algorithm is pure assembly code that can be compiled using the Atmel AVR compiler. Some modifications made it compilable using the `avr-gcc`. In addition the decryption functionality has been removed as it is not necessary for hash function constructions. If an AES decryption is needed, it can be easily added by the cost of a small increase in code size. Furthermore, we contributed our own implementation of the MixColumns function that is better in respect to performance and code size.

The used hash function constructions often apply the initialization vector as the encryption key. For a further speed-up we also implemented an alternative method with a pre-expanded key. This allows to save many key expansions in the process of creating one-block hashes.

Memory Management. The Merkle signature scheme is *key evolving*, which means that after every signing process a modification of the private key is required. The private key needs to be stored in nonfluent memory when the power is lost. Our implementation stores the private key in EEPROM, since the maximum amount of erase/write cycles allowed on the flash memory are usually much more limited than on the EEPROM. Our target platform is specified for at least 100.000 erase/write cycles [2,3]. Therefore the maximum value for the height of the Merkle tree supported by our implementation is $H = 16$, which allows $2^{16} = 65.536$ signatures to be generated. We store the whole signature in the SRAM during creation. However, for platforms that are even more constrained concerning SRAM our scheme can easily be altered to support signature generation and verification with much less RAM.

The memory constraints also enforce a very economical way of organizing the data of the private key. Despite of heavy optimizations, some implementation details force the actual size of the private key to be slightly larger than the

calculated results from Section 2.1. The main reason is that these formulas count only the number of hash values that must be stored. For example, the stacks used by the BDS algorithm were implemented as arrays of fixed size. In addition to the stack, we need to store the index of the array element that denotes the top node on the stack. Also the size of the signatures is slightly larger than estimated, because the index of the signature must be added as well.

Key Generation. Due to the heavy computations required, the key generation is not done on the microcontroller but on a standard PC. For the generation of test data, we created a PC version of the project that uses mostly the same code base. In contrast to the AVR implementation it uses a different implementation of the AES algorithm and it supports key generation. The key generation is computationally far too costly to run on the microcontroller which is why it has been disabled in the AVR implementation. The speed of the PC version has also been used to verify the correct behavior of our code by iterating through all possible signatures.

Side Channel Resistance. For smart card implementations, resistance against side channel attacks is of high importance. All parts of an implementation handling sensitive data need to be protected against a possible leakage. The W-OTS signature keys X and their seeds SEED are the only critical data. Since the keys X and their seeds are used as one-time keys, the values are being processed very few times, rendering non-template based attacks difficult. However, the analysis of the vulnerability against side channel attacks and, if necessary, the development of efficient countermeasures are an interesting field for future work.

4 Choice of Parameters and Timings

In this section we present the timings of our implementation and the exact memory requirements for the microcontroller. We also compare these values to implementations of state of the art signature schemes on the same microcontroller platform. For the height of the Merkle tree we chose $H = 16$ and $H = 10$ which allows 2^{16} and 2^{10} signatures to be generated with one key pair, respectively. The reason for the choice of $H = 16$ is, that 2^{16} is near to the maximum number of allowed write cycles for the EEPROM of the microcontroller [23]. The values for $H = 10$ were included to clarify the impact of the tree height on the signature generation time. For the Winternitz parameter w and the parameter K for the BDS algorithm, we tested three combinations $(w, K) = (2, 2), (2, 4), (4, 4)$. The value t , that denotes the number of 128-bit strings in the W-OTS signature key and the one-time signature, is $t = 133$ for $w = 2$ and $t = 67$ for $w = 4$. Table 2 summarizes the results. $s_{\text{pub}}, s_{\text{priv}}, s_{\text{sig}}$, and s_{ROM} denote the memory requirements for the public key, the private key, and the signature as well as the code size in bytes, respectively. t_{verify} and t_{signing} denote the average time in milliseconds required for verification and signature generation, respectively.

Table 2 shows that our implementation features smaller code size and smaller public keys. The above figures already include the code size of the hash function

Table 2. Timings and memory requirements of our implementation and comparison to state of the art signature schemes on the same platform

Scheme	Memory in bytes				Time in msec	
	s_{pub}	s_{priv}	s_{sig}	s_{ROM}	t_{verify}	t_{signing}
Our MSS-128 implementation using $H = 16$						
$(w, K) = (2, 2)$	16	1440	2350	6600	85	1230
$(w, K) = (2, 4)$	16	1472	2350	6600	85	1072
$(w, K) = (4, 4)$	16	1472	1330	6600	127	1665
Our MSS-128 implementation using $H = 10$						
$(w, K) = (2, 2)$	16	848	2290	6600	82	756
$(w, K) = (2, 4)$	16	876	2290	6600	82	598
$(w, K) = (4, 4)$	16	876	1234	6600	124	946
RSA-1024 [14]	131	128	128	7400	215	5495
RSA-2048 [14]	259	256	256	10600	970	41630
ECDSA-160 [10]	40	21	40	43200	423	423
ECDSA-160 [17]	40	21	40	17900	1218	1001

needed for digest generation and the AES engine. Also the signature verification times are faster than those of RSA and ECDSA. The signature generation time of our implementation is much faster than the RSA implementations and comparable to the ECDSA implementations. In case of $H = 10$ our implementation is even faster than the memory efficient ECDSA variant from [17]. The main drawbacks of the MSS are the large memory requirements for the signature and the private key. However, both the private key and the signature easily fit into the EEPROM and the SRAM of the Atmel, respectively.

We finally remark that our implementation provides a practical security of 128 bits and hence offers long term security until the year 2090 [16]. RSA with an 1024-bit modulus offers comparable symmetric security of only 72 bit, i.e. until the year 2006. The security of 2048-bit RSA is at most 95-bit, i.e. until the year 2040. ECDSA using 160-bit elliptic curves offers only 80 bit of security, i.e. until the year 2018. This shows that our implementation is not only very competitive to currently used schemes, but also offers higher practical security [16].

5 Hardware Accelerated AES

The most performance critical part of our MSS implementation is the AES based hash function. Hence, a natural approach to improve the scheme's overall performance is to accelerate the AES implementation. Many recent low-/mid- budget smart card processors offer hardware acceleration for symmetric encryption schemes like the AES. One publicly available platform offering an AES hardware acceleration is the Atmel ATxmega 128A1 processor.

In contrast to the software implementation of the SBL construction using AES that requires 4,081 cycles per block, the hardware accelerated version requires

only 1069 cycles per block. However, the AES engine itself needs only 375 cycles on the target platform. Besides the overhead for the hash function construction the remaining cycles are mainly required for the necessary memory management. Table 3 illustrates the performance of the MSS when utilizing AES hardware acceleration. This table shows, that speeding up the hash function by a factor ≈ 3.82 results in a speed up of the Merkle signature scheme by roughly the same factor.

Table 3. Timings and memory requirements of our implementation with and without AES hardware acceleration

Scheme	Memory in bytes				Time in msec	
	s_{pub}	s_{priv}	s_{sig}	s_{ROM}	t_{verify}	t_{signing}
Our MSS-128 implementation using $H = 16$ / Software AES						
$(w, K) = (2, 2)$	16	1440	2350	6600	85	1230
$(w, K) = (2, 4)$	16	1472	2350	6600	85	1072
$(w, K) = (4, 4)$	16	1472	1330	6600	127	1665
Our MSS-128 implementation using $H = 16$ / Hardware AES						
$(w, K) = (2, 2)$	16	1440	2350	6100	24	362
$(w, K) = (2, 4)$	16	1472	2350	6100	24	317
$(w, K) = (4, 4)$	16	1472	1330	6100	38	504

6 Conclusion

We presented an implementation of the Merkle signature scheme on a low-cost 8-bit microcontroller platform. Our implementation shows that MSS is a competitive signature scheme compared to commonly used signature schemes such as RSA and ECDSA. Our implementation has smaller code size and faster verification times than efficient implementations of RSA and ECDSA. The signature generation times are faster than RSA and comparable to ECDSA.

When employing an available symmetric crypto coprocessor, even further speed up can be reached.

Our implementation gives a positive answer to the question whether highly secure and efficient signature schemes can be implemented on constrained devices.

References

1. Rijndael/rivest implementation (January 2008), <http://point-at-infinity.org/avraes/>
2. Atmel. Overview of secure avr microcontrollers 8-/16-bit risc cpu (2007), <http://www.atmel.com/products/SecureAVR/>
3. Atmel. Specifications of the atmega128 microcontroller (2007), http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

4. Buchmann, J., Coronado, C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS - an improved merkle signature scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006)
5. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007)
6. Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited (manuscript, 2008), <http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/dahmen.html>
7. Cheng, X., Li, W., Znati, T. (eds.): WASA 2006. LNCS, vol. 4138. Springer, Heidelberg (2006)
8. Coronado, C.: On the security and the efficiency of the merkle signature scheme. Cryptology ePrint Archive, Report 2005/192 (2005), <http://eprint.iacr.org/>
9. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 96–115. Springer, Heidelberg (2005)
10. Driessen, B., Poschmann, A., Paar, C.: Comparison of Innovative Signature Algorithms for WSNs. In: Proceedings of the First ACM Conference on Wireless Network Security (to appear)
11. ePractice.eu. Belgian electronic ID card officially launched (April 2003), <http://www.epractice.eu/document/2139>
12. Digital signature standard (DSS). FIPS PUB 186-2 (2007), <http://csrc.nist.gov/publications/fips/>
13. Ganesan, P., Venugopalan, R., Peddabachagari, P., Dean, A., Mueller, F., Sichitiu, M.: Analyzing and modeling encryption overhead for sensor network nodes. In: WSNA 2003: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, pp. 151–159. ACM Press, New York (2003)
14. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and rsa on 8-bit cpus. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
15. Labor, D.: Crypto-avr-lib (January 2008), <http://www.das-labor.org/wiki/Crypto-avr-lib>
16. Lenstra, A.K.: Key lengths. Contribution to The Handbook of Information Security (2004), <http://cm.bell-labs.com/who/akl/key-lengths.pdf>
17. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. Technical Report TR-2007-36, North Carolina State University, Department of Computer Science (November 2007)
18. Luk, M., Perrig, A., Whillock, B.: Seven cardinal properties of sensor network broadcast authentication. In: Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks, pp. 147–156 (2006)
19. Menezes, A.J., Vanstone, S.A., van Oorschot, P.C.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
20. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
21. Naor, D., Shenhav, A., Wool, A.: One-time signatures revisited: Have they become practical. Cryptology ePrint Archive, Report 2005/442 (2005), <http://eprint.iacr.org/>
22. Steinberger, J.P.: The collision intractability of mdc-2 in the ideal-cipher model. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 34–51. Springer, Heidelberg (2007)

23. Viega, J.: The AHASH Mode of Operation (manuscript, 2004), <http://www.cryptobarn.com/>
24. Yu-long, S., Jian-feng, M., Qing-qi, P.: An Access Control Scheme in Wireless Sensor Networks. In: IFIP International Conference on Network and Parallel Computing Workshops, 2007, pp. 362–367 (2007)

Fraud Detection and Prevention in Smart Card Based Environments Using Artificial Intelligence

Wael William Zakhari Malek¹, Keith Mayes², and Kostas Markantonakis³

Royal Holloway, University of London
Egham, Surrey, TW20 0EX, United Kingdom
{Keith.Mayes,K.Markantonakis}@rhul.ac.uk
www.malek@gmail.com

Abstract. This paper discusses the development and research for the detection of fraud in Smart-Card environments by using artificial intelligence. The current research deals with behaviour based detection engine, which will detect abnormalities by learning the usual behaviour of the user and detecting new unusual behaviours. The behaviour-based detection engines is based on ‘Neural Networks’. This work considers the feasibility of implementing ‘Neural Network’ fraud engine on a Smart card platforms.

1 Introduction

There are many reasons for researching fraud detection mechanisms. Certain types of fraud are still very hard to detect by current technical security measures. Telecommunication and other Smart card based industries, such as payment cards, are still very vulnerable to fraud. The use of sophisticated fraud detection techniques can assist in early detection and prevention. Neural Networks and behaviour-based detection engines add true artificial intelligence to the security defences of the system, rather than the more conventional signature based security measures. With intelligent security in place, the development cycle of industry applications could go further and faster and take bold steps towards the future.

In researching this paper a Fraud Engine based on an Artificial Neural Network (ANN) was implemented on a Smart card in order to assess the performance and the general feasibility of this approach. The motivation was that, the intelligent behaviour-based security mechanisms can provide added protection for critical systems. Of particular interest is the real-time detection and reaction to fraudulent behaviours [1]. Any suspicious or unusual activities are captured and prevented instantly. With real artificial intelligence implemented using Neural Networks, behaviour based security mechanisms promise to be at the same step as the attacker and not a step behind. Using this kind of approach to security brings it down to the personal level, meaning fraud should be detectable for every single user or customer depending on his usage characteristics.

The Smart card environment is very limited, e.g. it has very limited memory and processing power and even modern Java Cards still have a restricted Java

Card Runtime Environment(JCRE). In this paper the possibility of using Artificial Neural Networks to detect fraud and unusual behaviour is investigated. The fraud engine implementation is tested on a PC environment and we will discuss the feasibility of the implementation on a Java Card. In the next section, we will discuss Neural Networks concept and how the brain works. In section 3, we will discuss the design and implementation challenges related to implementing an ANN on a smart card. In section 4, we will discuss the functionality of a fraud engine based on ANN. Finally, we will discuss the results and the benefits of using such a tool as well as the future suggested work.

2 Neural Networks Concepts

It can be argued that a behavioural based fraud detection engine protects and provides added security to Smart card based systems. It is considered necessary to add such a measure to security because Smart card applications such as banking cards and SIM/USIM mobile cards have become a crucial part of everyday life. Since, Smart Cards have become a part of our critical transactions, they can learn our habits and behaviours as we use them. A behavioural based security measure is an attractive method of protection but requires a sensitive but reliable detector. The proposed security mechanism is heavily dependent on Neural Networks and the degrees of intelligence they provide, as they learn the usual behaviour and are able to easily detect and stop the unusual ones. The idea behind the Neural Networks is to provide an artificial brain that will learn the same way that the human brain learns [2]. According to the required knowledge it will start to make decisions when anomalies are detected.

2.1 How the Brain Works

Before considering an artificially intelligent mechanism it is necessary to first understand a little about how the human brain works. The human brain has about 100 billion cells [3]. Most of these cells are called neurons. The interaction of multiple cells is called a neural network or biological neural network. A neuron is an on/off switch and is either in a resting state (off) or it is shooting an electrical impulse down a wire (on) [4]. It has a cell body, a long wire which is called an Axon, and at the very end it has a little part that shoots out a chemical. This chemical goes across a gap which is called Synapse where it triggers another neuron to send a message as shown in Figure 1 [5, 6]. There are many neurons sending messages using Axons to communicate with the neighboring cells [7]. As the message goes down an Axon to a neighboring cell it could pass through a lot of other cells on the way until it stops at a certain destination as shown in Figure 1. The establishment of this route, which starts from a starting cell to a destination cell, when activated is called learning [6]. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes [3].

According to the concept on how learning happens in the brain, scientists were able to create an Artificial Neural Network (ANN) based on the understanding

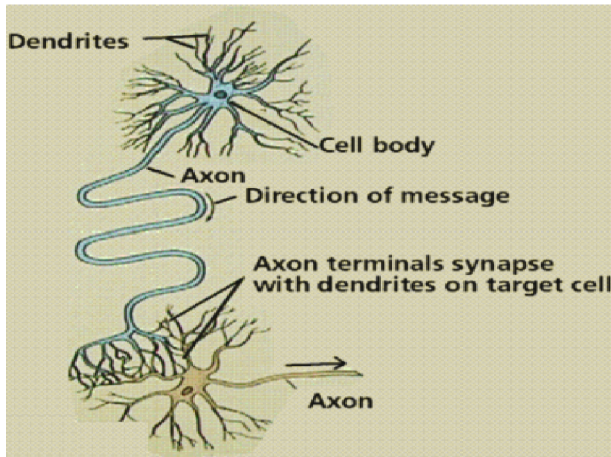


Fig. 1. Neuron Cell Connection [18]

of how the brain's Biological Neural Network works [5]. The aim was to simulate the human brain on an application and employ it to provide the advantages of the human brain and make it available to maximize the performance of systems and applications or even the actual computers.

Artificial Neural Networks are typically composed of interconnected units, similar to the brain. There are input units which serve as the dendrites in the brain [8]. The function of the synapse is modeled by a modifiable weight, which is associated with each connection between the dendrites (input units) and other units (other cells). Each unit converts the pattern of incoming input (experience) that it receives from the other units into a single outgoing activity that it broadcasts to all other units. It multiplies each incoming input by the weight of the connection and adds together all these weighted inputs to get a quantity called the total input [8] Figure 2.

The behaviour of an ANN depends on both the weights and the input-output function (transfer function) that is specified for the units. When the inputs are received through the input units, the output is calculated and all the weights in the middle, between the input units and the output units, are modified. Through this process learning happens. In other words, learning take place in an ANN by modifying the weights according to some learning mathematical calculations and functions, which take place at every unit. The behaviour of the output units depend heavily on the activity of the hidden units and the weights between the hidden and output units [9].

2.2 Artificial Neural Network Advantages

The fact that the ANN can detect patterns, identify familiar ones, as well as identify anomalies and give a best guess is a remarkable advantage over any other conventional design. In addition, the ANN will not need a constant increase in

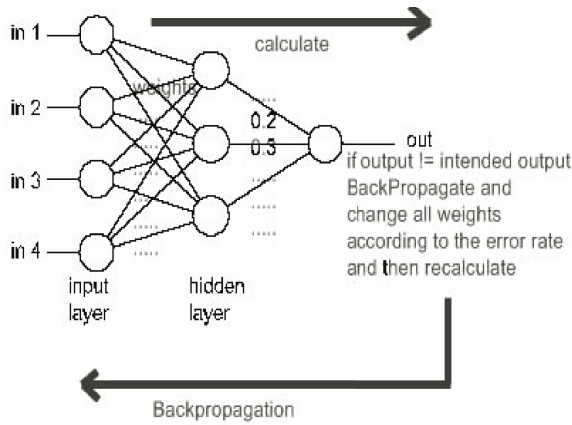


Fig. 2. Artificial Neural Network

size as it gets smarter, which makes it a unique approach to designing complex and intelligent applications. The smarter the human brain gets through education, the more memory it holds and the more information it contains, however the brain does not get bigger the more education we get. The same idea works for the ANN. We might decide to implement more hidden layers or hidden units but that does not happen as frequently as assigning new memory slots in a data base design approach. Another advantage is that, to calculate an output for a given input, the ANN goes through series of mathematical functions. These functions are heavily dependent on addition and subtraction in plus some simple multiplication or division that does not take a lot of processor time. In other words, it is fast to calculate and efficient, which makes it suitable for hardware implementation and restricted environments such as Smart Cards platforms.

2.3 Existing Artificial Neural Network Based Systems

There are already systems that are designed to analysis behaviour to detect potential fraud. Such systems, known as Fraud Engines, are based on studying a certain behaviour and reporting if a different behaviour is detected. Neural Fraud Management Systems (NFMS) that are completely automated and state-of-the-art integrated system of neural networks, Fraud Detection Engines and Automatic Modeling Systems [10, 11].

A machine learning, anomaly-detection ANN based system can be used to address the shortcomings of rule-based systems. Rather than having to wait for a new attack to be detected and for a new rule to be written by an expert, these systems automatically and immediately detect unusual behaviour for each user and for groups of users. Behavioural systems are inherently future proof as they can spot new types of attacks the first time that they are executed. An effective anomaly detection system relies on clustering algorithms that are based on Artificial Neural Networks. A clustering algorithm groups similar transactions

into a small number of clusters. Each cluster represents a common pattern of activity. Each time a new transaction is processed by the anomaly detection system, the system tries to fit it into an existing cluster. If a transaction does not fit into any cluster, it is classified as an anomaly [12].

3 Neural Networks and Smart Cards

The aim of this paper is to examine the feasibility of implementing a neural networks based fraud engine on the Smart card it-self and not on the centralised processing systems, such as Visa or the mobile phone authorisation systems. The fraud engine will be installed on the Smart card in order to evaluate the current card behaviour and analyse the card usage and authorise actions accordingly. The usage behaviour will be stored on the card. If unusual behaviour is detected the carrier device of the Smart card will ask the user a secret question that only the legitimate user can answer. If the secret question is answered properly the requested use will be permitted and the card's ANN will learn the new behaviour so the user will not be prompted again when performing a similar task. In the case of the user not being able to answer the security question, the requested use will not be allowed and the card will be temporarily locked. This implementation guarantees that only the Smart card holder is the one in full control of the card all the time, and if the card is to be stolen, the card cannot be used to commit fraudulent activities. Of course before this scenario can be considered we must first determine if it is indeed feasible to implement the ANN on a smart card.

In this section we review the limitations of the Smart card environment and examine the possibility of implementing an ANN on Smart card. on a Smart card and in particular a card platform supporting the Java Card Run Time Environment (JCRE).

3.1 Design Challenges

In this section we will introduce the challenges that faces the design and the implementation of an Artificial Neural Network on a smart card.

Data Structures. As mentioned earlier, to be able to successfully implement an ANN we will need to store the weights in a data structure that will facilitate access and modification of the weights. In the research we used multidimensional arrays to store the different sets of weights, but in JCRE multidimensional arrays are not supported, which means this data structure is no longer viable for use by the ANN tool. JCRE supports one dimension arrays which is the only data structure available and so a conversion from multidimensional arrays to one dimensional arrays is a must.

To be able to implement this conversion successfully the ANN tool's code need to be reviewed carefully to guarantee that the tool is accessing the correct weights at any point in time. The design of the ANN needs to be modified in order generate results as per the original design. In the next section we show the original code and the modified code to demonstrate how the conversion is possible as per Figure 4.

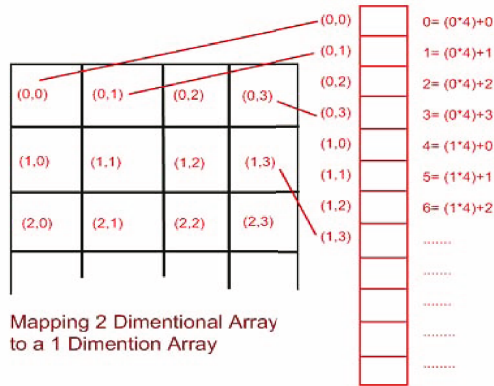


Fig. 3. Mapping two-dimensional array to one-dimensional array

As shown above, if the mapping process is implemented successfully, we can have an ANN implemented based on a single dimensional array. The access to the weights to the single dimensional array will need to be modified. For example, in the 'for loops' used to access and modify the weights instead of using a double index (2,3) we will only use a multiple of the row plus the column number ($i \times 2 + 3$), where i is the number of rows (Figure 4). By applying this formula we can overcome the first challenge and be able to access the weights correctly and modify them safely.

Complex Math. Due to the limited processing power of the Smart card's CPU, complex calculations are not recommended as they will be very time consuming and therefore impractical. In theory, only addition and subtraction take minimal time. Multiplication (which is simply a multiple addition) and division are very time-consuming, and should be avoided if possible. The manipulation of the weights is very simple; it is only a formula with simple math such as division and multiplication. However, the activation function which is used in clustering and mapping behaviours necessitates some demanding maths by using complex functions such as Sigmoid [8]. The Sigmoid function needs to calculate the exponentials of the weights.

The first attempt to solve this complex problem is to implement the exponential function by using simple math only which is possible if the concept of the exponential function is understood correctly. The exponential function, $EXP(x)$, is defined as the sum of the following infinite series [13]:

One obvious way of writing this program is computing each term directly using the formula $(x^i/i!)$. However, this is not best practice, since both x^i and $i!$ could get very large when x or i is large. One way to overcome this problem is rewriting the term [13] as shown in Figure 5.

Therefore, the $(i+1)$ th term is equal to the product of the i -th term and $x/(i+1)$ [13]. The second design makes it feasible to implement using a simple 'for loop' in addition to some simple math such as addition and multiplication. However, although this implementation was feasible for some computing platforms, it is not suitable for restricted environments such as Smart Cards. The

$$\begin{aligned} \exp(z) &= 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \dots + \frac{z^i}{i!} + \dots \\ &= 1 + \frac{z}{1} + \frac{z^2}{2!} + \frac{z^3}{3!} + \dots + \frac{z^i}{i!} + \dots \\ (i+1)^{th} \text{ term} &= \frac{z^{i+1}}{(i+1)!} = \frac{z \cdot z^i}{(i+1)i!} = \frac{z}{i+1} \cdot \frac{z^i}{i!} = \frac{z}{i+1} (i^{th} \text{ term}) \end{aligned}$$

Fig. 4. Implementing the Exponential Function [13]

computation time needed to calculate the right exponent is completely dependent on the number fed into the function and the tolerance level. If the tolerance level is low, the time needed to calculate EXP(x) is long, but if the tolerance level is high, the time is less, but the accuracy is poor.

Another proposal is to replace the exponential activation function by the identity function to simplify the procedure, but extra limits are then needed to be added to help the weights not to converge in magnitude and cause an error (Figure 6). As a solution, instead of using Sigmoid functions, which need complex calculations, we use a threshold function that keeps the weights within the required limits (Figure 6).

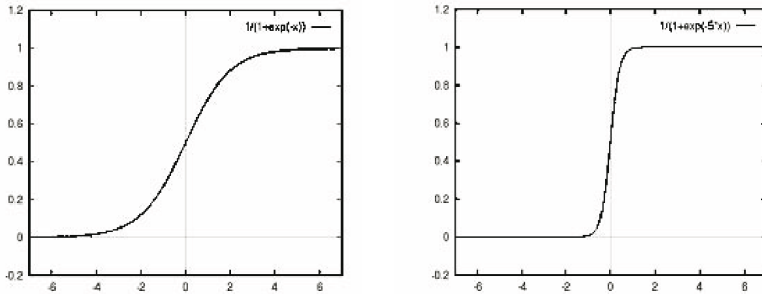


Fig. 5. Sigmoid Function graph [35], Threshold Step Function

By using the threshold step function we are able to comply and fall within the limitation of the Smart card environment and be able to come a step closer towards implementing a successful application that better cope with a Smart card.

3.2 Implementation Challenges

ANN with Integer Weights. The main challenge is that the Smart card environment does not support Float or Double types. Meaning that, it only supports

integer and Short types; types with a minimal precision. The entire ANN implementation had to rely on integer maths only. This is a major challenge in the design of the ANN that can affect the results dramatically. Selecting weight precision is one of the important choices when implementing ANN and may be used to trade-off the capabilities of the realised ANN against the implementation cost [14]. A higher weight precision means fewer quantisation errors, while a lower precision leads to simpler designs, greater speed and reductions in area requirements and power consumption, which is ideal for the Smart card environment [14]. By keeping the precision to a minimum, we will be improving efficiency and speed as well as complying with the power restrictions [14].

There has been extensive research in this area proposing advanced algorithms that use only neural networks with integer weights that produces similar accuracy with minimum error rates [15]. The majority of those algorithms use the negative of the gradient of the error function, $E(w)$, as their descent direction. The gradient $E(w)$ can be computed by the BackPropagation [16, 17] of the error through the layers of the network. This calculation, however, is computationally expensive and difficult to implement in hardware [16]. In this paper, a new class of ANN that do not need the floating point weights, has to be researched. Other algorithms, that train neural networks with threshold units, require the learning task to be static. However, in order to train the network 'off-line' in a software simulation and later transfer it to the hardware [18]. Card and so many real life applications may not be static, i.e., input data may continue to change even after the hardware implementation, which is the situation in the case of an ANN on a Smart card, the learning should take place as the user uses the Smart card. In such cases an algorithm capable of continuing training 'on-chip' is needed. There have been some proposed strategies that are capable of continuing the training process in hardware, when threshold activation functions have been used [18].

By using algorithms that only uses integer math in restricted environments such as Smart Cards, we guarantee speed and reduced space requirement. Integer math will result in a reduction in power consumption and maximising the limited performance. An ANN with only integer math will have integer weights only, which is proved to be faster and more accurate compared to networks with floating point weights [19]. However, using only integer math will require all values in the design to be integers such as the error rates and the weight change variables. If such variables are rounded up or down, this might increase or freeze the error rate which will cause inaccuracy in the calculations. By using fixed point math a number can be represented in two parts, an integer part and a fractional part.

Solution - Integer Math and Bit Manipulation. Every unsigned integer in the new proposed design is represented by 8 bits. The lowest order 3 bits represent the fractional part of a number and the remaining 5 bits represent the integer part, for example [19]:

$0 = 00000.000$, $2 = 00010.000$, $1.25 = 00001.010$

So that the full range of this number is:

00000.000 to $11111.111 = 0$ to 31.875

Now we can use normal integer arithmetic operations and bit shift the operands or result to acquire the correct answer.

For example [19],

Multiplication

$00001.010 * 00010.000$ ($1.25 * 2.0$) = 10100.000

The integer maths has [effectively] performed the multiplication with an extra factor of 2^3 or 1000 binary, which we can now adjust for by shifting the answer 3 bits to the right:

$10100.000 >> 3 = 00010.100$ (2.5 decimal). [19]

Division

$00001.010 / 00010.000$ ($1.25 / 2.0$)

If we wish to keep the same level of precision in the answer then we must first shift the numerator 3 bits to the left,

$00001.010 << 3 = 01010.000$

Now we can perform the integer division,

$01010.000 / 00010.000 = 00000.101$ (0.625). [19]

Addition / Subtraction

$00001.010 + 00010.000$ ($1.25 + 2$) = 00011.010 (3.25)

This is the correct answer. No manipulation required. [19]

The bit shifting technique is the ideal way of implementing ANN on a Smart card, as it will save both power and computation time. The bit shifting techniques could be incorporated and calculated in the activation function so we do not need to perform the bit shifts in the ANN actual calculations. From experimental results it has been proved that this design is actually slightly faster than any ANN using floating point weights [16], which is a great achievement that emphasises the idea of having an ANN on a Smart card.

4 The Fraud Engine

4.1 Design

The design of the fraud engine is based on a neural network with Back Propagation learning algorithm, where we modify the weights by the error rate produced from calculating the difference between the desired and the actual outputs. The neural network should receive some inputs and produce a decision (Approve/Reject), Figure 3. The inputs will depend on the use of the tool e.g. if it is used as a Telecommunication fraud engine, the inputs could be a phone number and a time of call, or duration of call and a destination country. If the fraud engine is used for a payment card, the inputs could be an amount of a

transaction and a time of transaction. The inputs to the fraud engine must define the user behaviour and be crucial to the purpose and use of the Smart card. The decision is based mainly on the ANN weights that have been learnt and optimised from previous. If the Smart card is used for the first time it will start recording the first few activities which will reflect its decision on the future ones.

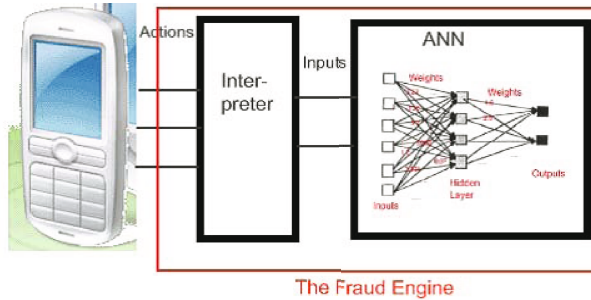


Fig. 6. Our proposed design of a fraud engine

4.2 How Will It Work?

The weights are originally randomly initialized and stored in a multidimensional array. The network has 3 layers i.e. the input layer where the inputs are fed to the network, the hidden layer where extra cells and neurons exist and the output layer which is the decision enforcement point. The inputs to the network are specified by the environment of the Smart card e.g. the telecommunication SIM cards might have inputs such as call information and financial Smart Cards such as credit cards might have transaction information as inputs. The inputs are fed to the network and the learning process starts. After the first few transactions the network starts to build an idea about the user and his behaviour which defines the thresholds for the user’s actions. As the learning process progresses the weights change and they have different magnitudes that reflect the network’s new state. Future behaviours are reflected by new inputs which are fed to the network, if the behaviour is recognized the output is positive and approval of the behaviour is guaranteed otherwise the user will be asked for security questions which he should answer correctly if he is the genuine owner of the Smart card at the time. If the answer is correct, the action is permitted and the new behaviour is learned, otherwise the action is prohibited.

4.3 The Fraud Engine Tool

In this example we will assume that this tool is implemented for use on a mobile phone to provide a behavioural based fraud detection engine. The inputs could be assigned by the manufacturer of the phone, but in this example we will use 2 inputs only; the time of the call and the call destination. Ideally the time of the call should be easily understood and interpreted by the Fraud Engine, but we

might have some complications with (AM/PM) timing. For simplicity we will divide the day into 4 hour intervals and we will refer to them as Time Zones and they will be numbered as follows:

```
12:00:00 PM until 04:00:00 PM: Zone 1
04:00:01 PM until 08:00:00 PM: Zone 2
08:00:01 PM until 12:00:00 AM: Zone 3
12:00:01 AM until 04:00:00 AM: Zone 4
04:00:01 AM until 08:00:00 AM: Zone 5
08:00:01 AM until 12:00:00 AM: Zone 6
```

The time zone will be one of the inputs to the fraud engine, the second input will be another number reflecting the destination of the call which will be called the Destination Number. It could be represented by another set of numbers which reflect the destination country or even the city. Assuming it will be ranging from (0..9), where 0, 1 are reserved for local calls and (2..5) for continental calls and (6..9) for international calls. When the user attempts to make a phone call, both the number called and the local time of the call are translated by the phone into the corresponding Destination Number and Time Zone which will be fed into the ANN for processing.

During the setup process of the phone with the new SIM Card, the phone should be able to store securely some information about the SIM Card holder such as an OIN, birth date, age, address, postal code, house number, a memorable word, a memorable day or any piece of information that the user could remember later on and be able to be authenticated to the phone or the SIM Card if requested. After the setup process, the information retrieved should be stored locally and securely on the mobile phone, Figure.

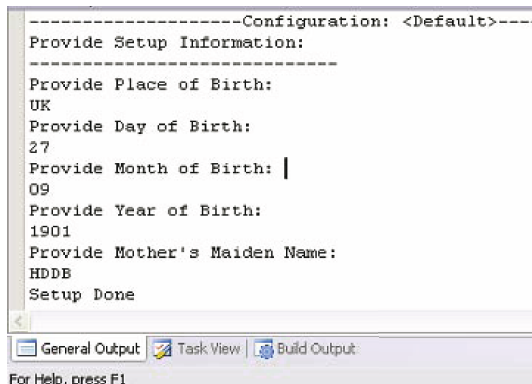


Fig. 7. Provide setup information, which is done only once at the beginning and the information is then kept secret

4.4 Scenarios

In the first scenario, we assume that the user of the mobile and the SIM Card make only local calls between 12:00 PM and 8:00 PM, which means the ANN is trained to accept and approve calls to local destinations on Time Zones 1 and 2 and 3. A call interpreter function translates a call action to two inputs; the destination (0,1) and the time zones (1,2,3) which represent the user’s behaviour. The fraud engine was tested by first making a similar call behaviour (Figure 3). Which means the interpreter will translate the similar call behaviour to the inputs (1,2) as for ”Local Call, at 5:30 PM”. In the second test scenario, we assume that the same user attempts to call China at 5:00AM interpreted by the input pair (8, 5), which is an unexpected behaviour. This anomaly will be detected by the fraud engine and it will not be approved as it is an unexpected behaviour that is not been learned by the ANN. In this case the user will be required to further authenticate himself and prove that he is the legitimate owner. For example, the user may be prompted for a random question based on the reference information stored earlier. If the user is able to give the correct answer the action or the call will be granted and the new behaviour will be learned, otherwise the card could be blocked or the user is transferred automatically to the customer service line Figure 3.

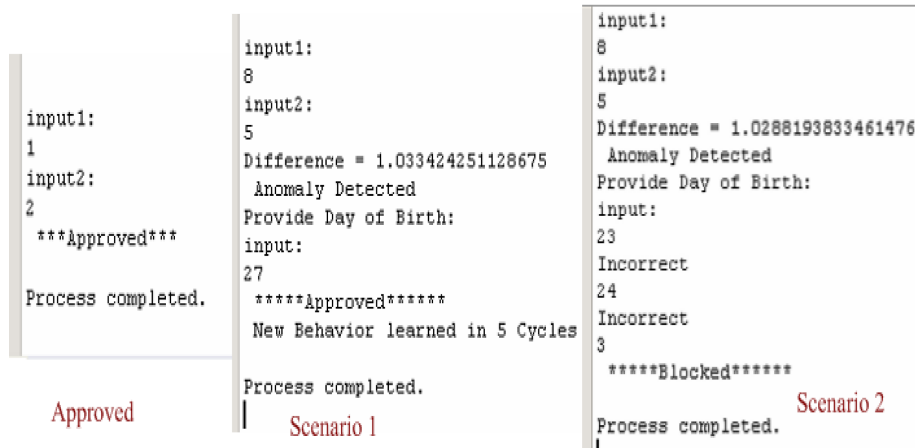


Fig. 8. Approved call or expected behaviour

In these two scenarios we were able to show how a behavioural based fraud engine can defeat fraud and recognise an unusual behaviour, which could help limit the use of stolen Smart Cards and provide a second wall of defence against stolen SIM Cards and mobile phones. As shown earlier it is quite effective and user-tolerant as well. As an enhancement, the security questions could be behaviour based, such as asking the user about the most dialed number, or the last dialed number, or the last call received. These questions provide greater security, but there could be greater inconvenience for the user. Extensive research

has to be carried out by the mobile phone manufacturers or the SIM card issuers to find the best questions that provide both convenience and good security.

Apart from the choice of good security questions there are other challenges that need to be addressed such as interoperability. If the SIM card is to be removed from a handset and placed in another, would all the previous behaviours be lost? The answer to this question would depend on the design of the fraud engine. As mentioned earlier, the weights are the heart of the ANN and they reflect the state of the artificial mind. If the weights are to be stored on the phone, all the information would be lost if the SIM card moves to another handset. The solution is for the weights to be stored on the SIM card itself so it is possible to move the SIM card to another handset with a compatible fraud engine that will retrieve the weights and continue to offer the same level of protection.

Another challenge is the tolerance level. Low tolerance from the fraud engine will trigger more security questions which might effect customer satisfaction and cause the customer to switch the tool off, but it will provide higher security. Conversely, high tolerance will achieve better customer satisfaction, but lower the security. The solution to this is to train the ANN offline with the worst case scenarios of fraud as well as to first train for the expected use depending on the purchased calling plan. This initial training will provide a basic protection level that the user will be able to develop and customize according to his own habits. This approach will be useful and effective both in decreasing the alerts and increasing security.

One of the most important challenges is to be able to correctly learn the user behaviour and be able to adapt to the user's life style changes. In modern phones the user can choose different profiles of use. Every profile has ring tones, wallpaper, calling properties such as barring and forwarding, favorite numbers, etc. Every profile may represent a different life style such as work, travel (where roaming should be considered as typical behaviour), weekends and personal use. For every profile there could be a different set of weights that represent and reflect the use of the SIM card. Protection could be extended to cover all the different sets of behaviours a user can perform. When the profile is changed the ANN saves the current weights in the previous profile's security domain and loads the new set of weights for the new profile.

5 Conclusion

5.1 Benefits

Issuers have systems that protect them against fraud for which they are liable but these measures may not adequately protect the customer and so added security mechanisms may be desirable. Building good security and protection around every Smart card will decrease the pressure on the Card Issuers of having to counter all problems with centralised security mechanisms. The benefit of the ANN Fraud Engine is to provide a customised security for every single user. It provides mechanisms that are designed to protect every user according to his habits and usage of the SIM Card.

5.2 Results

We used a Java card 2.1 compliment for testing the execution of the fraud engine. It was tested on a simulator machine implemented on a PC. Although there have been some technical difficulties with recording the execution time on the Smart card, an approximation of the relative execution times has been recorded on the simulator. By recording multiple runs and taking an average it was possible to get an estimate of the execution time. The execution time of the Fraud engine when making a decision is 4.24 ms, which was achieved by recording 500 runs at 2.12 seconds. The period of time needed for learning a new behaviour is longer than the time needed to calculate the output of the ANN(make a decision). For the ANN to learn a new behaviour it needs 37.54ms at each cycle. In other words, to calculate the output of the network and get the error rate and then back-propagate to modify the weights it takes 37.45ms. The learning process will need at least 5 cycles to modify the weights efficiently so this execution time needs to be multiplied by 5 to get the overall learning time which is 187.70. The learning process depends on the accuracy of the implementation which means the better the implementation the faster the execution time. One other major accomplishment, is the size of the applet implemented. the size of the applet was 879 bytes on the EEPROM.

5.3 Suggested Future Work

After discussing the previous challenges and the obstacles that could affect having an ANN based fraud engine on a Smart card, it was possible to find solutions to the major problems that face such an idea. After implementing and fine-tuning an ANN algorithm based on the previous findings, it was possible to test it on a Smart card platform. The results were positive in terms of speed and power consumption but with some degradation in accuracy. Therefore, More research is needed in the areas regarding the execution time and the bit shifting algorithm. There are some ways to improve the execution time. The first would be to test it on a real card because the simulator environment might not have given the best indication of the execution time. The host of the simulator machine had limited memory and other processing tasks, which means the Fraud Engine might actually perform faster on the actual Smart Card. By tuning the Algorithm of the ANN further, we might get a better execution time. Another important area to be researched is the Bit shifting algorithm. There is a current bug in the design causing the accuracy to be only 70% due to a flaw in the bit shifting algorithm explained in section 3.2, which means, further tuning of the bit shifting algorithm is needed in order to achieve better accuracy hence better results.

References

- [1] Harris, S.: CISSP, 3rd edn. Hardcover. Osborne (2005)
- [2] John, P., Jesan, D.M.L.: Human brain and neural network behavior: A comparison 1, 2-5 (2003) [accessed on 3/04/2007], <http://www.acm.org>

- [3] Dewri, R.: Evolutionary neural networks: Design methodologies 1, 1–5 (2003) [accessed on 9/02/2007], <http://aidepot.com/articles/evolutionary-neural-networks-design-methodologies/>
- [4] Davalo, E., Patrick Naim, A.R.: Neural Networks. MacMillan Education Limited, Basingstoke (1991)
- [5] Nicholls, J.G., Martin, A.R., Wallace, B.G., Fuchs, P.A.: From Neuron To Brain, 4th edn. Sinauer Associates Inc. (2001)
- [6] Stergiou, C.: Neural networks, the human brain and learning 1, 1–3 (1996) [accessed on 14/06/2007], <http://www-dse.doc.ic.ac.uk/nd/surprise96/journal/vol2/cs11/article2.html>
- [7] Khanna, T.: Foundations of Neural Networks. Addison-Wesley, Reading (1990)
- [8] Fausett, L.: Fundamentals of Neural Networks. Prentice-Hall, Englewood Cliffs (1994)
- [9] Omid Omidvar, J.D.: Neural Networks and Pattern Recognition. Academic Press, London (1998)
- [10] Dimension, N.: Fraud detection using neural networks and sentinel solutions (smartsoft). 1, 1–3 (2006) [accessed on 17/06/2007], <http://www.nd.com/resources/smartsoft.html>
- [11] Khan, A.: Feedforward Neural Networks with Constrained Weights. PhD thesis, Univ. of Warwick, Dept. of Engineering (1996)
- [12] VeriSign: Verisign identity protection fraud detection service an overview. Whitepaper, VeriSign (2006)
- [13] University, M.: Exponential function 1, 1–3 (2004) (12/07/2007), <http://www.cs.mtu.edu/shene/COURSES/cs201/NOTES/chap04/exp.html>
- [14] Vassilis, P., Plagianakos, M.N.V.: Parallel evolutionary training algorithms for hardware-friendly neural networks. Technical report, University of Patras, GR-26110 Patras, Greece (2002)
- [15] Pavlidis, N.G., Tasoulis, D.K., Plagianauos, V.P., Nikifuridis, G., Vrahatis, M.N.: Spiking neural network training using evolutionary algorithms. Technical report, 3Department of Pathology, University Hospital, GR26500 Patras, Greece (2001)
- [16] Vassilis, P., Plagianakos, M.N.V.: Training neural networks with threshold activation functions and constrained integer weights. Technical report, University of Patras, GR-26500, Greece (2001)
- [17] Magoulas, G.D., Plagianakos, V.P., Vrahatis, M.N.: Hybrid methods using evolutionary algorithms for online training. Technical report, Department of Information Systems and Computing, Brunel University, Uxbridge UB8 3PH (2001)
- [18] Sutton, J.Z.P.: Fpga implementations of neural networks - a survey of a decade of progress. Technical report, University of Queensland, Brisbane, Queensland 4072, Australia (2004)
- [19] SharpNeat-Developers: An integer based neural network, sharpneat.sourceforge.net. 1, 1–5 (2004), <http://www.sharpneat.sourceforge.net/integernetwork.html>

The Trusted Execution Module: Commodity General-Purpose Trusted Computing

Victor Costan, Luis F.G. Sarmenta, Marten van Dijk,
and Srinivas Devadas

MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, USA
victor@costan.us, {lfgs,marten,devadas}@mit.edu

Abstract. This paper introduces the Trusted Execution Module (TEM); a high-level specification for a commodity chip that can execute user-supplied procedures in a trusted environment. The TEM is capable of securely executing partially-encrypted procedures/closures expressing arbitrary computation. These closures can be generated by any (potentially untrusted) party who knows the TEM's public encryption key. Compared to a conventional smartcard, which is typically used by pre-programming a limited set of domain- or application- specific commands onto the smartcard, and compared to the Trusted Platform Module (TPM), which is limited to a fixed set of cryptographic functions that cannot be combined to provide general-purpose trusted computing, the TEM is significantly more flexible. Yet we present a working implementation using existing inexpensive Javacard smartcards that does not require any export-restricted technology. The TEM's design enables a new style of programming, which in turn enables new applications. We show that the TEM's guarantees of secure execution enable exciting applications that include, but are not limited to, mobile agents, peer-to-peer multiplayer online games, and anonymous offline payments.

1 Introduction

The Trusted Execution Module (TEM) is a Trusted Computing Base (TCB) designed for the low-resource environments of inexpensive commercially-available secure chips. The TEM can securely execute small computations expressed as partially-encrypted compiled closures. The TEM guarantees the confidentiality and integrity of both the computation process, and the information it consumes and produces. The TEM's guarantees hold even if the compiled closure author and the TEM owner do not trust each other. That is, the TEM will protect the closure's integrity and confidentiality against attacks by its owner, and will protect itself against attacks by malicious closure authors. The TEM does not trust the authors of the programs it runs. A malicious closure cannot negatively impact the TEM it runs on, and it cannot maliciously interfere with the result of closures written by other authors. This implies that there is no need for a program certification system.

The TEM executes compiled closures in sequential order, in a tamper-resistant environment. The execution environment offered by the TEM consists of a virtual machine interpreter with a stack-based instruction set, and a single flat memory space that contains executable instructions and temporary variables. The environment is augmented with a cryptographic engine providing standard primitives and secure key storage, and with a persistent store designed to guarantee the integrity and confidentiality of the variables whose values must persist across closure executions. The persistent store is designed to use external untrusted memory, so its capacity is not limited by the small amounts of trusted memory available on inexpensive secure hardware.

The TEM's design focuses on offering elegance and simplicity to the software developer (the closure author). The instruction set is small and consistent, the memory model is easy to understand, and the persistent store has the minimal interface of an associative memory. The breakthrough provided by the TEM is the capability to execute user-provided procedures in a trusted environment, for the low price of a commodity chip.

We have implemented the TEM on an inexpensive, commercially-available JavaCard. The TEM's prototype implementation shows that the design is practical and economical. The research code implements a full stack of TEM software: firmware for the smart card, a Ruby extension for accessing PC/SC smart card readers, a TEM driver, and demo software that uses the driver. The prototype implementation leverages the advanced features of the Ruby language to provide a state of the art assembler which makes writing compiled closures for the TEM very convenient.

The TEM enables new applications by combining the flexibility of a virtual machine guaranteeing trusted execution with the pervasiveness of inexpensive secure chips. For example, the TEM can be used to provide provably secure, simple, and general solutions to mobile agents, peer-to-peer multiplayer online games, and anonymous offline payments.

The outline of this paper is as follows. We start with related work in section 2 which compares TEM with existing approaches towards trusted computing. The main concepts used in TEM are introduced in section 3. Section 4 details the architecture of TEM and section 5 discusses its implementation together with timing results from experiments. Section 6 explains how to program closures that can securely migrate between TEMs. This mechanism can be used in applications such as secure mobile agents and secure peer-to-peer multiplayer online games. For a more detailed discussion on the TEM and downloadable source code, the reader is referred to the first author's Master's thesis [1].

2 Landscape

Large TCBs: Trusted Modules and Processors. Early solutions to secure platforms were supplied, most notably by IBM, as tamper-resistant assemblies that can operate either autonomously or as coprocessors for high-end systems. A recent representative of these systems is the IBM 4764 co-processor [2], which is only available for IBM servers under custom contracts.

Secure processors represent a different approach to trusted platforms. A secure processor costs less than a trusted platform because the secure envelope only contains the logic found inside CPUs. The AEGIS [3] design provides a cost-effective method for implementing a secure processor (that embeds an integrity checking interface to untrusted memory). Compared to the smart card chips, secure processors deliver higher performance and readily support large applications, but are much more expensive.

Embedded TCBs: Smart cards. Smart cards [4] are secure platforms embedded in thumb-sized chips. For handling convenience, the chips are usually embedded in plastic sheets that have the same dimensions as credit cards. The same chip are used as Subscriber Identity Modules (SIMs) in GSM cellphones. Smart cards have become pervasive, by offering a secure platform at a low cost.

The ISO 7816 standard regulates the low-level aspects of smart cards [5]. Platforms such as MultOS [6] and JavaCard [7] provide a common infrastructure for speeding up application development, and allow multiple applications from different vendors to coexist securely. However, both of these platforms were intended for monolithic applications that are contained and executed completely on the smartcard. Applications on a card can only be installed or updated if they are certified by a trusted entity, which is either the smart-card emitter, or the platform vendor.

The TEM design makes large applications easier to develop, because it loads one closure at a time into the secure environment, as opposed to smart cards that load all the applications at once. The TEM makes update deployment easier, because open classes are naturally implemented with closures. A TEM is also more flexible, as it allows its owner to execute any vendor's applications.

Fixed-Function TCBs: TPMs. The TPM is a fixed-function unit, which means it defines a limited set of entities (such as shielded locations holding cryptographic keys or hashes), as well as a closed set of operations that can be performed with primitives (such as using a key to unwrap another key or to sign a piece of data). The operations are not sufficient for performing arbitrary computation on the TPM. Instead, the TPM was envisioned to attest that the host it is attached to, a general-purpose computer, runs a trusted software stack. The strength of the bond between the TEM and its host determines the security of the entire system, since an attacker that compromises the bond can spoof the attestation system [8], [9]. The TCB on the TPM's host computer includes a secure boot loader, an operating system, and drivers. Such a TCB does not exist, because it is impractical to analyze and certify the large codebases of modern operating systems, together with their frequent updates. In practice, TPM applications (e.g., [10]) do not assume a TCB on the host. Thus, they are not capable of performing trusted arbitrary computation.

The TEM does not require trusted software on its host to perform arbitrary computation, and does not need to be securely bound to its host. This means that a TEM can cost less than a TPM, and that existing computers can be enhanced with TEMs via standard extension buses, like the USB.

3 Concepts

3.1 Trust Chain

The method used to attest that a platform offers the security guarantees of a TEM is a simplification of the TPM's chain of trust for platform attestation [11]. The root of trust is the hardware manufacturer (such as Infineon or Atmel), which acts as a Certificate Authority in a public key infrastructure as defined in [12]. Each TEM has a unique asymmetric key. The private part of the key (PrivEK - Private Endorsement Key) is generated inside the TEM at manufacturing time, and never leaves the TEM. The public part (PubEK) is included in an Endorsement Certificate (ECert) issued by the TEM's manufacturer, attesting that PubEK corresponds to a TEM endorsement key. Since a TEM has exactly one PubEK, the PubEK can be used to identify and track the TEM, and thus its owner. This may be unacceptable in some circumstances, as it leaks information about the users' identity. By adapting the ideas of [11] to augment the chain of trust, the TEM can be made untraceable (see [1]).

3.2 Closures

The closure is the execution primitive of the TEM. This allows the use of virtually any programming paradigm with the TEM. Compiled closures (described below) can be implemented in an execution engine that is just a bit more complex than an engine designed for procedural execution. This translates into a small¹ execution engine that is suitable for implementation on embedded platforms.

A *closure* (originally defined in [14]) is a fragment of executable code, together with the bindings of the variables that were in scope when the closure was defined. Closures are extremely powerful, and can be used to implement most primitive structures in modern programming languages ([15] and [16]). To provide immediate assurance to readers, listing 1 uses the approach employed by ECMAScript [17] (also known as JavaScript) to implement Object-Oriented Programming [18] objects featuring encapsulation.

```

1  def bank_account(account_number)
2    balance = 0
3    account = Hash.new
4    account[:balance] = lambda { balance }
5    account[:number] = lambda { account_number }
6    account[:deposit] = lambda { |amount| balance += amount }
7    account[:withdraw] = lambda { |amount| balance -= amount }
8  end

```

Listing 1. Bank Account object implemented with closures (functional Ruby)

¹ As compared to the Java Virtual Machine [13].

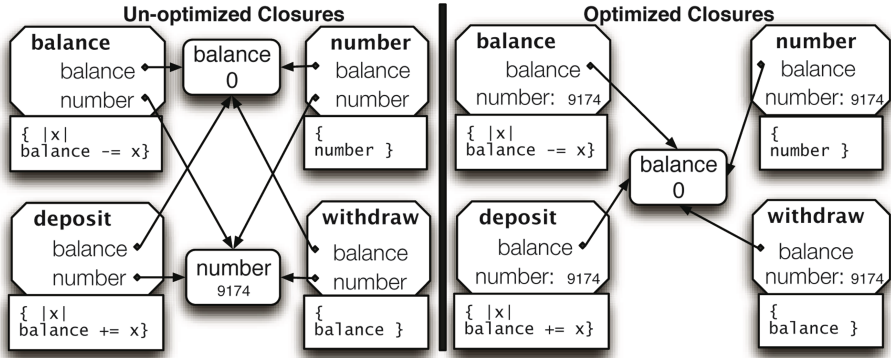


Fig. 1. The structure of the closures in the Bank Account object. Left: the straightforward result. Right: result after de-facto immutable variable optimization.

Figure 1 shows the structure of the closures created by executing listing 1. Each object method becomes a closure that contains a sequence of executable code, and a binding table that associates variable names with pointers to memory cells storing the variables' values.

Compiled Closures. The TEM design targets embedded chips, where persistent variables are expensive². The following optimization, inspired by [19], helps to reduce the amount of shared memory cells used by a closure. Some of the variable bindings are de-facto immutable (constant). That is, their values will never be modified throughout the lifetime of the closure. Thus, the constant value can be stored directly in each closure's binding table. [19] uses this mechanism to decide whether frames will be allocated on the stack or in the heap.

For example, it is easy to see that `number` in listing 1 is de-facto immutable, and `balance` is not. So the closures' binding tables can be optimized to use one shared memory location instead of two, as illustrated in Fig. 1.

The result in the right of Fig. 1 is further amenable to well-known optimizations, such as removing unreferenced variables from the binding table. For instance, the variable `number` is not used at all in the closures `balance`, `deposit`, and `withdraw`, so it can be removed from their binding tables.

A **compiled closure** is a closure that has been fully optimized for the computer that is intended to execute it. A compiled closure consists of the following:

- the computation to be performed, expressed as executable instructions that can be interpreted by the target computer,
- a binding table that contains all the non-local variables,
- values for the non-local variables that are de-facto immutable, and
- pointers to the shared memory locations holding mutable non-local variables.

² Variables' values may change, therefore they would have to be stored in EEPROM. EEPROM is the slowest and most expensive type of on-chip memory.

SEClosures and SECpacks. A **SEClosure** (Security-Enhanced Closure) is a closure where all the information has been classified as one of the following.

- **shared:** information whose integrity must be guaranteed by the TEM. For example, executable code requires integrity in order to detect whether it has been replaced by malicious code that would output private information.
- **private:** information that requires confidentiality guarantees from the TEM, such as a signing key or a secret algorithm. The integrity of private information must be protected by the TEM as well. Otherwise, the TEM owner could learn private information by executing closures where the private information was modified, and observing the differences in results.
- **open:** this information is not covered by any guarantee. This has to be information that the TEM’s owner supplies, as the owner is the only one who does not need any proof of integrity or confidentiality.

For simplicity, it seems appealing to remove the *shared* class of information, and specify that all the information not provided by the closure’s owner is private. However, SEClosures need to have *shared* information to allow the TEM owner to assert certain facts about the computation expressed in the closure.

SEClosures are compiled into a format that the TEM can easily process. A compiled SEClosure, called a **SECpack**, has all the shared, private, and open data grouped together.

Bound SECpacks. Before a SECpack is given to the TEM’s owner, its content is partially encrypted with the TEM’s PubEK, to protect the private and shared information. The encryption result is a **bound SECpack**, containing the same information as the original SECpack, but in a form that enforces the confidentiality and integrity of the enclosed information. A *bound* SECpack can only be executed by a platform possessing the PrivEK corresponding to the PubEK used for encryption.

Binding (explained below and summarized in Fig. 2) assumes that the TEM’s PubEK is known and was validated against the TEM’s ECert. The scheme is inspired by the TPM [11].

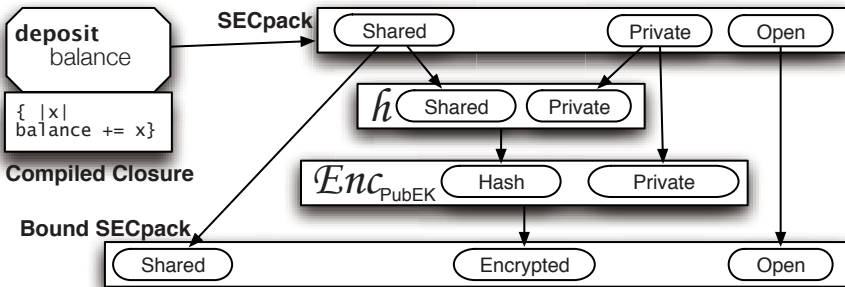


Fig. 2. The SECpack binding process

Binding is performed using the following steps:

1. Let \mathcal{P} be the private data, \mathcal{S} be the shared data, and \mathcal{O} be the open data.
2. Use a cryptographic hashing function h (e.g., SHA1 [20]) to compute a digest $\mathcal{H} = h(\mathcal{P}||\mathcal{S})$ of the concatenation of the private and the shared information.
3. Use the TEM's Public Endorsement Key to encrypt the private information together with the digest; $\mathcal{E} = \text{Enc}_{\text{PubEK}}(\mathcal{P}||\mathcal{H})$.
4. The bound SECPack is the concatenation $(\mathcal{S}||\mathcal{E}||\mathcal{O})$ of the encryption result \mathcal{E} , the shared information \mathcal{S} , and the open information \mathcal{O} .

3.3 Persistent Store

The values of mutable variables are stored inside a secured global persistent store (Fig. 3), indexed by addresses that are at least as large as cryptographic hashes. An address identifies a value, and at the same time shows proof of authorization to access that value. The information in the persistent store is stored in a way that prevents any accesses that would bypass the associative memory abstraction. The values stored in the persistent store have the same size as the addresses, to avoid memory waste. The associations can be stored in untrusted memory on the TEM's host, using the architecture in section 4.3.

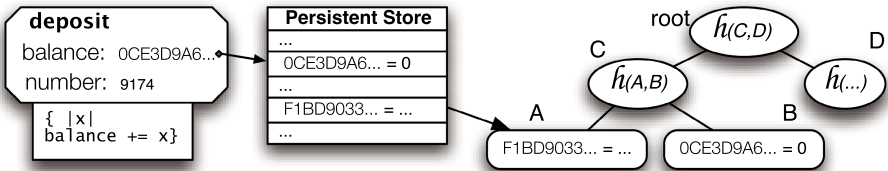


Fig. 3. Closure referencing the persistent store, with external memory tree

Assigning Persistent Store Addresses. A mutable variable uses the same persistent store address on all the TEMs it exists on. Addresses are assigned during closure compilation, using a random number generator. This simplifies deployment, because updates to a system can be easily implemented as new closures that use existing variables (in OOP, this is called open classes). Closures can also be easily migrated by re-binding the SECPacks to a different PubEK.

The probability of two different variables colliding on a TEM is at least as low as for Universally Unique IDs [21]. The probability of an attacker compromising a variable by guessing its address is at least as low as the probability that the attacker will be able to forge the signature on an Endorsement Certificate and break the chain of trust directly.

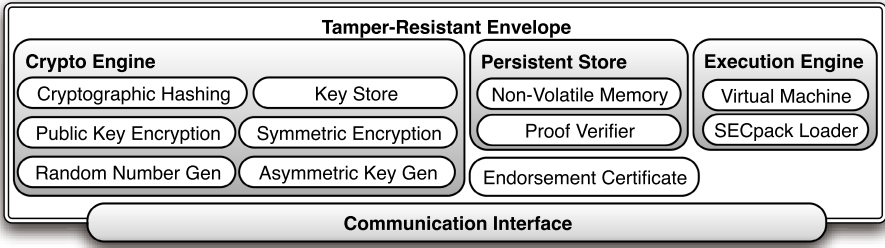


Fig. 4. High-Level TEM Block Diagram

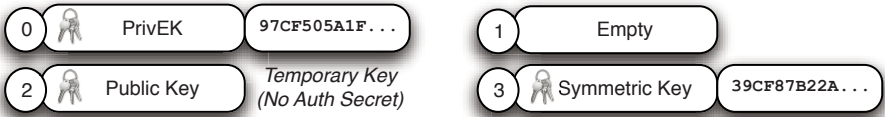


Fig. 5. Snapshot of a TEM's Key Store

4 Architecture

The main components of the TEM architecture (block diagram in Fig. 4) are the execution engine, the cryptographic engine, and the persistent store.

The TEM also contains a communication interface that is mainly a transceiver for the communication channel between the TEM and its owner. If the communication between the TEM and its owner occurs via an untrusted channel, the interface establishes a secure session using a mechanism similar to SSL [22], and relying on the TEM's ECert and PubEK to bootstrap the session.

4.1 Key Store

The key store (illustrated in Fig. 5) implements secure key storage, accessible as an array of key slots. A slot can contain a symmetric key, or the public or private part of an asymmetric key. This was inspired from the `javacardx.crypto` API [7], and chosen because each slot has well-defined `encrypt` and `decrypt` operations.

A key created during SEClosure execution is temporary by default, and is released when the closure ends executing. A key becomes persistent when it is given an authorization secret. Secrets have the same size as a cryptographic hash. A closure gains access to a persistent key by presenting the associated secret.

PrivEK occupies the first slot in a TEM's key store, and is associated with an authorization value known only by the platform manufacturer. The manufacturer can build "privileged" SEClosures, which use a TEM's PrivEK to offer functionality on top of the basic model, such as TPM emulation.

The TEM owner can remove any key from the store, via driver commands. This prevents the possibility of denial of service by filling up the key store.

4.2 Virtual Machine Environment

The computation inside a SECPack is expressed as microinstructions for a **stack-based virtual machine** (VM). The entire VM interpreter state consists of an instruction pointer (IP) and a stack pointer (SP). Instructions are encoded as a 1-byte operation code (opcode), optionally followed by immediate data. The stack consists of machine word-sized entries.

Using a virtual machine makes the executable code in a SECPack universal. The alternative of having SECPack executable code target specific hardware would introduce complexity (multiple target platforms) for the closure compiler, and would greatly complicate migrating SECPacks among TEMs.

The performance degradation introduced by the virtual machine has no significant impact on most TEM applications. Assuming a reasonable VM implementation, the time cost of one asymmetric key encryption operation dwarfs the cost of interpreting thousands of VM instructions. Asymmetric key decryption is invoked for every bound SECPack, because a part of the SECPack must be decrypted with the TEM's Private Endorsement Key.

Stack-based instruction sets are easy to interpret and generate from ASTs (Abstract Syntax Trees), and are used in recent VMs for both medium- and high-level languages (e.g., the Java VM [13], the Ruby 1.9 VM [23]).

The TEM's execution environment has a **single, flat, RAM-backed memory space** that contains executable instructions, values of local and de-facto immutable non-local variables, and the virtual machine's stack. Closures have direct access to the memory space, for maximum flexibility and performance (e.g., self-modifying code).

Closures return data by placing it in the **output buffer**, an append-only memory zone. If a closure's execution is aborted, the output buffer is discarded and nothing is returned. This mechanism simplifies building closures that don't leak information.

The VM contains special-purpose instructions accessing the functions of the cryptographic engine. A closure is automatically authorized to use all the encryption keys that it creates. The SEClosure can also use keys that are already loaded in the cryptographic engine, once it demonstrates knowledge of their authorization secret. The execution engine enforces these restrictions, and aborts closures that attempt to use a key before gaining authorization.

The contents of mutable non-local variables are stored in the persistent store (sections 3.3 and 4.3) between executions. Addresses are stored in memory space, and values are transferred between the memory space and the persistent store. If a closure is aborted, all its persistent store updates are rolled back.

The execution engine design completely discards any possibility of concurrent execution. This makes security guarantees easier to prove. A multi-core TEM can be modeled as multiple execution engines that share a persistent store where each SECPack execution is treated as a transaction.

SECPack Contents and Loading. A SECPack consists of a snapshot of the initial state of the virtual machine's memory space, together with a header containing a magic value, the initial IP and SP values, and the sizes of \mathcal{S} and \mathcal{P}

(needed to decrypt a bound SECPack). Unbound SECPacks have an empty \mathcal{P} . This makes virtual machine setup trivial, given an unbound SECPack.

A bound SECPack requires that the loader decrypts the private information \mathcal{P} and verifies the integrity of the shared information \mathcal{S} . SECPacks that fail the integrity check $\mathcal{H} = h(\mathcal{S}||\mathcal{P})$ are rejected.

The loader can use any key in the store (section 4.1) to decrypt a bound SECPack. This allows for speed optimizations and extensions to the TEM’s chain of trust. For example, an often-used set of SECPacks can be bound using a symmetric key instead of PubEK, if the key is somehow transmitted securely to the TEM. This can dramatically reduce execution time by avoiding asymmetric-key operations when SECPacks are loaded.

Considerations in the Design of the Instruction Set. The standard instructions are heavily inspired by the Java Virtual Machine [13]. The TEM-specific instructions (cryptography and persistent store) have been developed while aiming to adhere to the same principles.

The instruction set tries to strike a balance between enabling small SECPacks and keeping the VM interpreter simple. For example, most instructions operating on memory blocks have two variants. The fixed block variant (instructions ending in `fb`) receives the information about the blocks (address, and optionally length) as immediate data. The variable block variants (instructions ending in `vb`) pop the block information off the stack. The fixed block variant takes up less space in a secpack, while the variable block form provides maximum efficiency when working with variable-length memory structures.

Exceptions were made for instructions that would not occur often in a SECPack (e.g., `rnd`), so the space savings do not warrant the extra complexity in the interpreter. The instruction set aims for consistency with respect to mnemonics and order of parameters.

4.3 Persistent Store Architecture

Backing the Persistent Store by Untrusted Memory. Like AEGIS [3], the persistent store relies on building a Merkle tree [24] (Fig. 3) where the leaves store the actual associations, and internal nodes store a cryptographic hash of their children. The TEM stores the tree’s root in NVRAM but all the other nodes can be stored in untrusted memory. A symmetric encryption key³ inside the TEM⁴ is used to encrypt the two parts (address and value) of each association individually, so neither part is stored “in the clear” in untrusted memory, and the TEM can later ask for an association by its encrypted address. The internal nodes hash the external representation of their children.

The TEM’s host maintains the tree structure. When a closure `reads` from the persistent store, the TEM communicates the encrypted address to the host. The host responds with the encrypted value at the address, and a correctness proof

³ If the law does not allow symmetric encryption, an asymmetric key can be used instead, at a large performance cost.

⁴ This key never leaves the TEM, and can be generated cheaply by a PUF [25].

consisting of all the nodes on the path to the leaf. `writes` work similarly, but the correctness proof also describes the updates to be performed on the tree. Rolling back an aborted closure is done by rolling back the tree root on the TEM.

The external memory tree exhibits **amnesia** if TEM’s host, who is managing the tree, can lie by telling the TEM that no association exists for an address, when in fact an association does exist. Undetected amnesia during a `read` can lead a SEClosure to assume that a variable has never been assigned on the TEM, and that it has its default value. During a `write`, undetected amnesia can lead the persistent store to create a duplicate leaf for an association, instead of updating the existing one. The TEM’s host can then return the (old) duplicate leaf for `reads`, effectively making the persistent store “forget” updates to a variable.

Amnesia is **detectable** if the lies of the TEM’s host can be recognized and stopped from propagating into incorrect SEClosure execution. For example, amnesia is detectable if, for every persistent store operation, it is known in advance if the persistent store already contains an association for the requested address.

If amnesia is not detectable, the correctness proofs given by the TEM’s host must ensure that amnesia cannot occur. In particular, the host must be able to prove efficiently that the tree does not contain any occurrence of an address. This requires external trees that are sub-optimal for a sparse address space (e.g., trees with a fixed branching factor), or complex to implement (e.g., binary search trees).

In comparison, if amnesia can be recognized, the TEM’s host does not need to supply any proof when it states that an address does not exist in a tree. The requirement that a proof is given when addresses are found is sufficient to ensure correct operation under detectable amnesia. So the external tree structure can be chosen in a way that maximizes the efficiency and simplicity of the proof validating process that is performed by the TEM.

The Lifetime of Persistent Store Variables. In order to avoid a complex tree structure that may require non-trivial resources, SEClosures must manage the lifetime of their persistent store variables. The scheme must detect amnesia in the external tree, and also be able to distinguish between the case when a variable has never been used on a TEM, and the case when the variable has been assigned a value, but the corresponding association has been **removed** from the persistent store. The possibility of confusing the two cases can be exploited by replay attacks. The method below achieves these requirements.

Let an **object**⁵ be a group of SEClosures that use the same mutable non-local variables. For convenience, an object’s **fields** shall be all the mutable non-local variables used by the SEClosures in the object. For the example in section 3.2, an individual bank account is an object consisting of the SEClosures `withdraw`, `deposit`, `balance`, and `number`. The object has one field, the variable `balance`.

To reduce complexity, all the fields of an object are managed as a whole, following the same principles as constructors and destructors (also named finalizers). Namely, an object is **constructed** on a TEM by creating persistent store

⁵ Object-Oriented Programming is used to simplify the presentation. However, the mechanism presented here can be easily adapted to other programming paradigms.

associations for all its fields. An object is **destroyed** by removing the persistent store associations. An object's SEClosures abort execution if any of the fields they reference do not have a value in the persistent store, so the SEClosures are only usable between the object's construction and destruction.

The process uses a single monotonic counter, \mathcal{M}_C , that is a mutable variable for the privileged SEClosures involved in object construction. The TEM owner is given a SEClosure that signs \mathcal{M}_C with PrivEK, so the read can be verified by anyone who has the TEM's ECert.

The object's owner receives a \mathcal{M}_C read result, and constructs the **constructor data** for an object, which consists of the \mathcal{M}_C value and the object's **constructor table**, a list of persistent store addresses and initial values for the object's fields. The constructor data is encrypted with the TEM's PubEK then given to the TEM's owner, together with the bound SECpacks for the object.

The owner runs the **constructor**, a privileged SEClosure that decrypts the construction data, and creates the associations in the **constructor table** if the \mathcal{M}_C value matches. If the object is constructed, \mathcal{M}_C is incremented. This way, exactly one object is constructed for a certain value of \mathcal{M}_C , and no constructor table is executed twice. This ensures that an object can be constructed at most once on a TEM.

The owner removes the object's fields from the persistent store by using the object's constructor data with a privileged SEClosure called the **desstructor**.

5 Implementation

The **TEM firmware** was implemented on JavaCard [7] smart cards, because of their widespread availability. The firmware's overall design closely reflects the TEM architecture illustrated in Fig. 4. Each component is materialized in one class with **static** fields and methods. The implementation manages its own memory buffers, to make optimum usage of the RAM and EEPROM memory on the chip, and to overcome the 255-byte limitation in APDU size on the prototype cards. The VM interpreter is implemented as one single 420-line method, and takes heavy advantage of the consistency in the instruction set.

The **TEM driver and SDK** were implemented in Ruby. Domain-Specific Languages (DSLs) [26] [27] were built for the TEM's data types and VM instructions. The SECpack assembler is also a DSL supporting comments, multiple instructions per line, named parameters, named labels, human-readable immediates, embedded Ruby code, as illustrated in listing 2.

The prototype TEM implementation contains supplemental features to help debugging SEClosures, such as dumping the TEM state when SEClosure is aborted. The SECpack assembler provides line-level debug information that is combined with the TEM state to pinpoint the exact line in the SECpack source code that is causing the execution to be aborted.

```

1 def gen_bank_account(number, initial_balance, bank_key)
2   balance_address = (0...ps_addr_length).map {|i| rand(256)}
3   balance_length = 8
4
5   balance_sec = assemble do |s|
6     s.psrdfxb :addr => :balance_addr, :to => :balance
7     s.ldwc :bank_key; s.rdk
8     # will output balance + signature(nonce + balance)
9     s.dup; s.ldkl; s.ldwc balance_length; s.add; s.outnew
10    s.outfxb :from => :balance, :length => balance_length
11    s.ksfxb :from => :nonce, :length => ps_value_length +
12          balance_length, :to => 0xffff
13    s.halt
14
15    s.label :bank_key; s.immed :ubyte, bank_key.to_tem_key
16    s.label :balance_addr; s.immed :ubyte, balance_address
17    s.label :nonce; s.filler :ubyte, ps_value_length
18    s.label :balance_value; s.filler :ubyte, ps_value_length
19    s.stack; s.extra 8
20  end
21 end # ( code for other closures and for the constructor)

```

Listing 2. Assembly code for `balance` in the bank account example

The TEM driver contains a full⁶ suite of unit tests, covering both driver and firmware code. The unit tests automate validating both the driver and the firmware, as well as assessing the suitability of a JavaCard model as a TEM.

5.1 Performance Considerations

The table below shows the results of various tests that were run on two JavaCard models. The times are expressed in seconds. Each time is the average over multiple consecutive repetitions of an operation. The number of repetitions was chosen such that the results of running an experiment (all the consecutive repetitions of the operation) 3 times were within 1% of the mean.

Operations	NXP 41/72k	Philips 21 18k
Process APDU	0.0061 s	0.029 s
Create and release 512-byte buffer	0.084 s	0.98 s
Decrypt with PrivEK	0.76 s	1.60 s
Execute 1-op SECPack	0.16 s	0.50 s
Execute 1020-ops SECPack	0.82 s	1.99 s
Execute 1-op bound SECPack	0.89 s	1.92 s
Execute 1020-ops bound SECPack	1.53 s	3.07 s

⁶ rcov indicates a line coverage of 95% or above on each Ruby source file.

The prototype’s performance is not yet acceptable for interactive systems. This is mainly because the VM interpreter is layered on top of the JavaCard virtual machine, which introduces the overhead of interpreted versus native execution (likely on the order of 20X). In practice, the performance hit is not as dramatic (a full 20X), because a significant part of a SEClosure’s execution time is spent on RSA operations on the TPM-grade 2048-bit keys. Cryptographic operations are done in hardware and do not incur the JavaCard overhead.

6 Example Application: Migratable Tokens

This section discusses the use of the TEM’s architecture to implement migratable tokens (as defined in [10]) in an anonymous offline payments system. The technique can be reused in other applications, such as authorization tokens in personal DRM, or active items (e.g., spells) in peer-to-peer online games.

A bank emits a sum of e-money as a TEM object (section 4.3). The object contains the bank’s private key, and the sum of money it represents. An object’s balance is verified by executing the `balance` SEClosure that produces a signature for the current sum of money, together with a caller-supplied nonce.

Making a payment is achieved by creating a money object on the receiver’s TEM. This is achieved in two steps:

- the SECpacks are migrated by a privileged SEClosure that verifies the receiver TEM’s ECert, then binds each SECpack to the receiver TEM using its PubEK.
- the balance is migrated by a `transfer` SEClosure that also verifies the receiver TEM’s ECert, then subtracts the required amount from the balance on the payer’s TEM, and produces constructor data that will create the required balance on the destination TEM. If the payer’s TEM balance reaches 0, the variables of the money object are removed.

Merchants use the same migration procedure to transfer the money to the bank. The bank then uses a `cancel` SEClosure that destroys the money object, and thus cancels the e-money it has emitted.

This scheme assumes that e-money never reaches the same TEM twice, and does not support merging e-money objects on a TEM. These restrictions can be removed by using a more complex scheme that is outside the scope of this paper.

7 Conclusion and Future Work

This paper introduces a novel approach to trusted execution. The TEM enables a new style of programming, by being capable of executing untrusted closures in a secure environment.

Understanding the applications of the TEM’s execution model is a fruitful avenue for exploration. Flexible trusted execution at commodity prices should bring new life to difficult problems, such as secure mobile agents.

Improving the TEM's performance is a promising prospect, as the benchmarks in section 5.1 are close to the results needed for interactive systems. The prototype JavaCard implementation can be optimized by using code generation or bytecode generation techniques. Assuming an adequate SDK, the TEM can be implemented directly on a secure chip, and will show the needed improvements.

The TEM's design can also be modified to fit specific goals or target platforms. There are many variations on the persistent store design. The execution environment allows extensions such as parallel processing on the crypto engine.

The TEM can be easily extended to offer certified closure execution. SECPack binding assures the closure's author that a closure was executed in a secure environment. Certified execution can prove to anyone that a result was produced in the secure environment of a TEM. We are currently investigating the applications of extending the TEM with certified execution.

References

1. Costan, V.: A commodity trusted computing module. Master's thesis, Massachusetts Institute of Technology (2008), <http://tem.rubyforge.org>
2. Arnold, T., Van Doorn, L.: The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. IBM Journal of Research and Development 48, 475–487 (2004)
3. Suh, G., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th annual international conference on Supercomputing, pp. 160–171 (2003)
4. Hendry, M.: Smart Card Security and Applications. Artech House (2001)
5. Husemann, D.: Standards in the smart card world. Computer Networks 36, 473–487 (2001)
6. Maosco, L.: (MultOS), <http://www.multos.com/> [cited May, 2008]
7. Sun Microsystems, I.: Java Card Platform Specification 2.2.1 (2003), <http://java.sun.com/javacard/specs.html> [cited May, 2008]
8. Lawson, N.: TPM hardware attacks. root labs rdist (2007), <http://rdist.root.org/2007/07/16/tpm-hardware-attacks/> [cited May, 2008]
9. Lawson, N.: TPM hardware attacks (part 2). root labs rdist (2007), <http://rdist.root.org/2007/07/17/tpm-hardware-attacks-part-2/> [cited May, 2008]
10. Sarmenta, L., van Dijk, M., O'Donnell, C., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In: Proceedings of the first ACM workshop on Scalable trusted computing, pp. 27–42 (2006)
11. Group, T.C.: Trusted platform module main (2007), <https://www.trustedcomputinggroup.org/specs/TPM/> [cited May, 2008]
12. Housley, R., Polk, W., Ford, W., Solo, D.: Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile (2002)
13. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc, Boston (1999)
14. Sussman, G.J.: Guy Lewis Steele, J.: Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab (1975)
15. Guy Lewis Steele, J.: Lambda: The ultimate declarative. Technical Report AI Lab Memo AIM-379, MIT AI Lab (1976)

16. Guy Lewis Steele, J., Sussman, G.J.: Lambda: The ultimate imperative. Technical Report AI Lab Memo AIM-353, MIT AI Lab (1976)
17. For Standardizing Information, E.A., Systems, C.: 262: ECMAScript Language Specification. ECMA, Geneva, Switzerland, third edition (1999), <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>
18. Cox, B.: Object oriented programming: an evolutionary approach. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
19. Guy Lewis Steele, J.: Rabbit: A compiler for scheme. Master's thesis, MIT AI Lab (1978)
20. Eastlake, D., Jones, P.: RFC 3174: US Secure Hash Algorithm 1 (SHA1). Internet RFCs (2001)
21. Leach, P., Mealling, M., Salz, R.: RFC 4122: A Universally Unique Identifier (UUID) URN Namespace (2005)
22. Freier, A., Karlton, P., Kocher, P.: Secure Socket Layer 3.0. IETF draft (1996)
23. Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter. In: Conference on OOP Systems Languages and Applications, pp. 158–159 (2005)
24. Merkle, R.: Protocols for public key cryptosystems. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 122–133 (1980)
25. Gassend, B., Clarke, D., van Dijk, M., Devadas, S.: Silicon physical random functions. In: Proceedings of the 9th ACM conference on Computer and communications security, pp. 148–160 (2002)
26. Cuadrado, J., Molina, J.: Building Domain-Specific Languages for Model-Driven Development. *Software*, IEEE 24, 48–55 (2007)
27. Cunningham, H.: Reflexive metaprogramming in Ruby: tutorial presentation. *Journal of Computing Sciences in Colleges* 22, 145–146 (2007)

Management of Multiple Cards in NFC-Devices

Gerald Madlmayr¹, Oliver Dillinger¹, Josef Langer¹, and Josef Scharinger²

¹ FH OOE F&E GmbH, NFC-Research Lab
Softwarepark 11, 4232 Hagenberg, Austria
{gmadlmay, odilling, jlanger}@fh-hagenberg.at
<http://www.nfc-research.at>
² Johannes Kepler Universität
Altenbergerstrasse 48, 4020 Linz, Austria
josef.scharinger@jku.at

Abstract. Near Field Communication (NFC) currently is one of the most promising technologies in handsets for contactless applications like ticketing or payment. These applications require a secure store for keeping sensitive data. Combining NFC with integrated smartcard chips in a mobile device allows the emulation of different cards. Representing each secure element with different UIDs poses several problems. Thus we propose an approach with a fixed UID dedicated to a Secure Element Controller (SEC). This approach allows an optimized backwards compatibility to already established reader infrastructures but also the communication in peer-to-peer mode with other NFC devices. Additionally the communication over peer-to-peer as well as the internal mode of secure elements at the same time is possible. This approach poses a flexible alternative to the implementations proposed so far. In addition when there are to multiple, removable secure elements in a device it is ensured that the secure elements are only used by authorized user/devices. The SEC in this case handles the communication between the secure elements as well as their authentication.

1 Introduction

Near Field Communication (NFC) is a wireless communication technology to exchange data up to a distance of 10 cm. NFC uses inductive coupled devices operating at the frequency of 13.56 MHz. NFC, standardized in ISO 18092, ECMA 340 and ETSI TS 102 190, is compatible to the standard of contactless smartcards (ISO 14443). An NFC device can operate either in active mode (generating a field and initiating the communication) or in passive mode (unpowered, waiting for a request) [1]. An NFC device is typically made up of two parts: the NFC part responsible for peer-to-peer communication, and a secure element storing sensitive data. Traditionally they are treated as separate devices, represented by different UIDs. But this approach poses several problems.

Despite of possessing multiple IDs, an NFC device has only one antenna. This resource must be shared which by mutually routing antenna I/O to the NFC-IC and the secure elements. As a consequence, a polling device can easily get

confused: Which secure element and which NFC-ID belong together? Polling also takes longer as multiple cycles are needed to fetch all identifiers.

From a technical point of view it is possible to mix both data streams and thus allowing simultaneous access. However, certain existing RFID infrastructures which NFC should be compatible with do not allow more than one device in range (e.g. MasterCard's PayPass [2]) because of security concerns. Many applications communicate via NFC but need some of the secure element's capabilities at the same time (e.g. storage for certificates, encryption of peer-to-peer communication). Using security features of the secure element is not possible with current architectures [3] (Fig.: 3(b)). If one device is selected no other data transfers are allowed. Only after the active NFC connection has been released the secure element can get selected.

For an application it would not be acceptable to interrupt the NFC session for every request that needs to be sent to the secure element. After every break it would be necessary to do a new polling cycle as the target must change its random UID for each new selection. An NFC device should support multiple secure elements. For example, it may have a Universal Integrated Circuit Card (UICC) but may also allow SDIO expansion cards to be inserted. Now the initiator has several problems: Which of the listed ID's are secure elements? Which do belong together?

In our proposal these problems are solved by assigning every NFC device an own UID. The unique ID is complementary to the already existing random 0x08-range ID. When acting as a target, the device will only expose its ID and appears as a legacy contactless smartcard. But there is a difference when getting selected: The device will set bit 6 in its SAK reply, telling its peer it is NFCIP-1 enabled. A legacy infrastructure will ignore this bit and further believe talking to a smartcard. All requests sent will be processed by a Secure Element Controller (SEC) which will distribute the data appropriately. This way it is possible to use many different interfaces for secure elements: conventional T=0/1, upcoming USB, SDIO, SWP, etc.

An NFC device wanting to set up a peer-to-peer connection will just continue sending an ATR request after enumeration. The SEC will intercept this package and divert it to the NFC core. From that time on the device runs in NFC peer-to-peer mode. Another possibility would be to require the initiator to send a special APDU which would cause the target to switch into NFC target mode and reply with its 0x08 random address. The initiator could then immediately select the target without a prior request.

That way, an application can also exchange data via peer-to-peer and send internal requests to a secure element the same time, as the wireless connection is not needed. This data path will be routed through the controller, made available to applications by Host Controller Interface (HCI) commands [4], for example. The HCI defines the architecture of the secure elements around the NFC controller and also specifies communication between secure elements. Thus an internal secure element could communicate with an external tag (e. g. in a smartposter) or establish a communication with another secure element. This

possibility could be used to implement an authentication mechanism between different secure elements. Authentication is a necessity to prevent fraud as a secure element should only work in authorized handsets.

The following section provides an insight into NFC technology and necessary background information for our idea proposed. Section three deals with the problem statement this paper is dedicated to. The concept of the single ID Secure Element Controller will be discussed in section four. The paper ends with a conclusion.

2 Near Field Communication

NFC is an amendment to already existing contactless smartcard technologies, while still being compatible to them. The integration of the technology in devices is driven by the NFC Forum, an industrial consortium founded by Nokia, Sony and NXP. The major idea was the creation of a general proximity communication standard to bridge the gap between Sony's Felica and NXP's Mifare Technology but also other smartcard standards based on ISO 14443 (A and B). Both, Mifare and Felica are proprietary implementations of contactless memory cards based on ISO 14443-3. NFC itself is defined in ISO 18092 and uses ISO 14443-A's data encoding scheme for transfer rates of 106 kBit/s and Felica for higher speeds (currently rates of 212 kBit/s and 424 kBit/s are defined, but may go up to 3 MBit/s in the future). NFC devices are able to interact with existing RFID readers by emulating cards (proximity inductive coupling cards, PICC). To read contactless smartcards or RFID tags, NFC devices can act as a reader or writer (proximity coupling device, PCD), respectively [5]. These two operation modes, card emulation as well as the reader/writer mode of NFC, are necessary for the compatibility with existing proximity card infrastructures.

Additionally, two NFC devices can exchange data through the peer-to-peer mode. Similar to PICC/PCD mode one device is the initiator (master) whereas the other device acts as a target (slave). The communication flow is initiated and controlled by the master device while the target only sends replies. There are two different modes. Either only the initiator emits a field in order to send requests while the targets answers by load modulation using the field or both devices create a field alternately [5]. Communication links between more than two participants at a time are currently not supported.

Although NFC technology is *work in progress* the major parts for device integration are covered by standards. There is an interface standardized between the host controller as well as the NFC controller, named Host Controller Interface (HCI). The HCI allows to control the NFC controller as well as to communicate with internal secure elements [6] and external targets or initiators. The HCI also defines an interface between the NFC controller and multiple secure elements. There can be various types of physical connections between the NFC controller and the SE, such as sign-in/sign-out (S2C) or Single Wire Protocol (SWP) for example [7]. The SWP is a special protocol that allows the NFC controller to exchange data with a UICC attached (Fig.: [8]).

Per default, any NFC device is in target mode and thus does not generate an RF field and waits silently for a command from the initiator. In case an application on

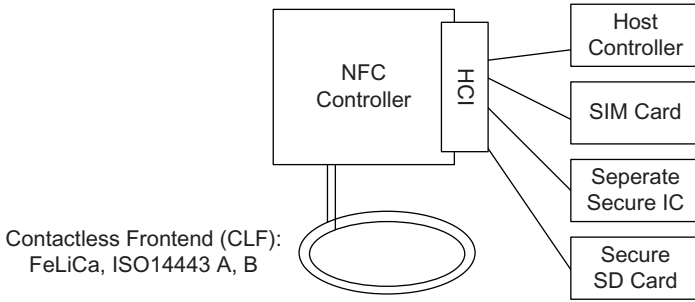


Fig. 1. NFC integration architecture using the HCI

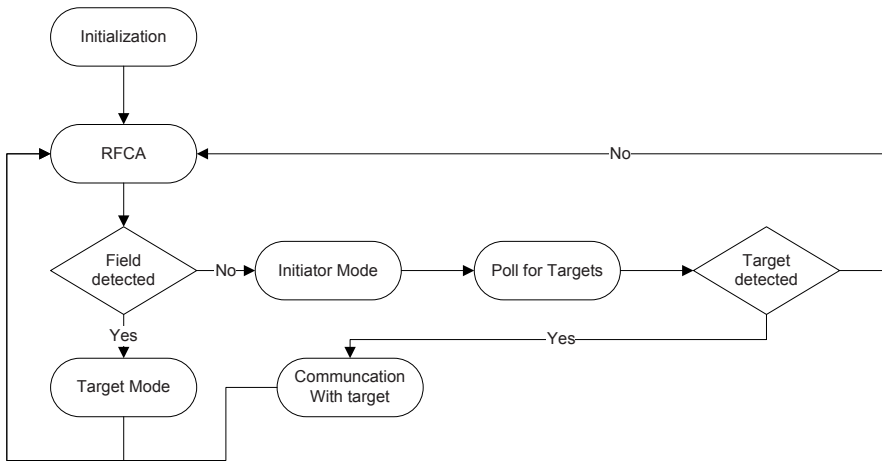


Fig. 2. NFC Mode Switch (simplified)

the device requires the initiator mode, the devices can switch into this mode. The field for the initiator mode is only activated, in case no external field is detected. In order to prevent two or more initiators to generate a field at the same time, a RF collision avoidance (RFCA) is implemented in each device. If there is no external field detected the device establishes a field, polls for an NFC device in range and initializes the communication with the target [1].

Whereas in classic smartcard environments the roles of the PCD and the PICC are clear, this is not the case for NFC devices as the conditions are different. As every device can initiate a communication, act as a peer-to-peer target or a transponder, a method is required to avoid chaos if several devices are close together. The approach of implementing the so called *Polling Loop* or *Mode Switch* is discussed in several papers as well as patents [8], [9], [10] (Fig.: 2).

While the device is in target mode, an external reader will detect two (or more) devices represented by UIDs: an NFC target as well as emulated cards. According to ISO 14443 and ISO 18092, targets can be represented by a 4, 7 or

10 byte long unique ID. Whereas the smartcard chip usually uses a fixed ID, the NFCID1 starts with 0x08 while the bytes left are random. This approach was chosen in order to protect the privacy of the device holder.

3 Problem Statement

The issue with the existing implementation, as lined out in the previous chapter, is that a reader will detect multiple UIDs of one NFC device. The implementation causes the following problem:

Such an NFC device will not work with legacy readers which only allow one transponder in the field of the reader (mono mode card readers). These kinds of readers do not implement or do not make use of the ISO 14443 anti collision and therefore are not able to establish a communication in case there are multiple targets in the field. Regarding NFC devices, the reader infrastructure needs to be replaced in order to be compatible with the new devices. For this case, the backward compatibility is not given. For example, the Nokia 3220 shows multiple UIDs (secure element and NFC) during the polling loop.

Even if the reader is able to process multiple cards in the field, already existing readers are not aware of the fact that UIDs starting with 0x08 are NFCID1s. Thus the reader will select each identifier in the field and setup a communication channel with the target. As the HCI allows the attachment of several secure elements to the NFC controller, this process is inefficient and time consuming. NFC readers do not have this problem, as long as they want to establish a peer-to-peer connection. In this case the NFCID1 is selected. With regard to selecting the correct secure element/smartcard, also NFC readers face the same problem as ordinary contactless smartcard readers, as they can not know which UID to select. We will deal with this issue in the first part of the paper.


Additionally, having one or more removable secure elements other than the UICC requires additional mechanisms against theft, abuse and management. For example, secure SD Cards containing contactless applications could be taken from the original device and be inserted into a new NFC device. The application provider who has data stored in this SD Card will not be informed about this issue. Thus locking the secure element or managing contactless applications over-the-air (OTA) can not be granted as the application provider does not know the new Mobile Station International Subscriber Directory Number (MSISDN) of the new handset. This problem will be discussed in the second part of the paper.

3.1 Possible Solutions

Explicit Card Select: In this case the user has to explicitly select one of the secure elements to be presented to the reader. From a technical perspective the implementation is simple and will solve the problem in an efficient way. From a usability perspective it is not satisfying. One reason is that the user has to know which secure element to choose, as there can be multiple secure elements in the device. Additionally it is not possible to simply browse the

secure element in order to figure out, which applications there are on which smartcard chip. This is due to privacy issues, because any other party would be able to see all the applications installed in the secure element as well. Secondly the explicit select is a complex process for the end-user from an interaction point of view. And this is actually not the idea behind NFC. The *Touch and Go* philosophy is a clear statement for a simple interaction between devices without browsing menus. Even if the explicit selection of the secure element is not an issue (in case there is only one SE in the device), the device still might operate in NFC target mode and thus send an NFCID1. Thus the user would directly have to select the application to use (payment, Bluetooth pairing etc.).

Time Multiplex: By using a time multiplex the secure elements are presented consecutively to the reader. In this case the reader will only see one UID at a time. The process is the following. The reader selects the first card and tries to access the application required. If the application is not found, the secure element is put into halt state. The halt state is the signal for the NFC device that the secure element selected was not the correct one. Then the reader sends another ISO 14443 request command. As a reaction the NFC device will present the second secure element to the reader. This process is repeated as long as the reader has found the secure element/NFC target required. If the required smartcard chip is not found and all cards of the NFC device are in halt state, no more card would be presented to the reader. From a technical point of view this is a feasible solution. The issue with this implementation is, that the reader has to put the card into halt state in case it was not the transponder it was looking for. This is actually not yet the case for reader infrastructures deployed and would require software/firmware updates of these readers. From a usability point of view, the implementation is user friendly, as there is no interaction required and the user does not have to care about switching between different modes or applications. A minor issue is the fact, that in worst case the reader has to go through all different cards/NFC targets in a device, which is not efficient.

Aggregation with one single UID: The third option would be the use of only one UID per device. In this case the NFC controller has to route the commands accordingly to the secure elements. This requires additional software running in the NFC controller inspecting if possible the data sent between NFC target/secure elements and external reader. From the technical side this implementation is the most complicated one. However, it neither requires any user interaction nor any modification to an already existing infrastructure. The major disadvantage of this approach is, that applications using the UID for identification or cryptographic functions (e. g. Mifare ) will not work. But as the UID of a smartcard is public using it for critical tasks as already mentioned is unsafe anyway. From the current perspective using only one UID per device is a suitable solution that allows interoperability with any established infrastructure. Therefore we consider this option for our implementation.

4 Implementation

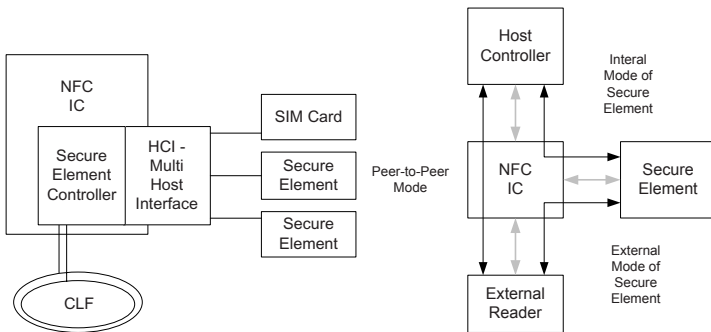
We propose the use of only one UID per NFC device and integrate a Secure Element Controller (SEC) to internally route the communication flows (Fig.: 3(a)). The SEC is a software implementation running in the NFC controller. For communication with the different instances such as secure elements, external readers or the host controller Application Protocol Data Units (APDU) are used. The communication handling makes use of the underlying HCI. The NFC device will be given an identifier out of the 0x08 range in order to correctly represent an NFCID1. NFC peer-to-peer capabilities are indicated by bit six in the SAK/SEL response of the NFC device after the selection command [12], [1]. Bit five indicates whether there is an ATS available and further ISO 14443-4 commands are accepted. Even if this bit is set, proprietary protocols could be used on top. For example, this is also the case for JavaCards integrating a Mifare section. With regard to the Mode Switch, complexity is reduced. There will be only a switch between initiator mode and *one* target mode. The SAK in this case is accomplished by the SEC. The explicit tasks for the SEC are:

1. Acting as a proxy between the smartcard chips and external readers to route the data correctly.
2. Managing the communication between the secure elements to implement an authentication mechanism.
3. Direct the OTA management connections from the UICC to other secure elements.

4.1 External Communication

With regard to the external communication, the NFC device has to reply to the following requests:

NFC initiator: The initiator establishes an RF field, runs the select process and after the SAK received, the initiator sends the attribute request and will



(a) Integration of a Secure Element Controller (SEC). (b) NFC Operating Modes [3]

continue with NFCIP-1 frames. Already out of the data encoding scheme the SEC can assume the following: If it is ISO 14443 A (106 kBit/s), the device polling is an NFC initiator or a ISO 14443 A reader. In case the SEC receives a request using the Felica encoding scheme (212 or 424 kBit/s) the initiator could also be a Felica reader. As after the selection the initiator would send an attribute request as mandatory in NFCIP-1. In this case the SEC forwards the data stream to the NFC core indicating that the data stream is a peer-to-peer connection.

Smartcard Reader: To handle multiple, different smartcards, the SEC is required to keep track of the secure elements in the NFC device. When a secure element is inserted, the SEC analyses the type (Felica, ISO 14443 A or ISO 14443 B). The internal routing of the data stream is quite simple as long as there is only one instance of a smartcard type in the device. If there is one smartcard chip of a type, the routing gets more complex (Felcia is not considered at the moment, as there was too little documentation available).

With regard to Mifare, the SEC has to provide an aggregation of all the Mifare Application Directories (MAD) ID. Although the proprietary authentication and encryption of Mifare also uses the UID of the smartcard chip, and thus is not feasible with our implementation, this might be different for the upcoming Mifare+ using AES. The external Mifare reader in this case is able read the MAD from the SEC like from an ordinary Mifare card. When trying the access a special block containing the application data, the SEC redirects the authenticate-, read-, and write-command to this chip.

In JavaCard OS using GlobalPlatform the functionality is a little different. An external reader is not able to query all the applications available, without proper access rights. Usually the reader sends a select request with the Application Identifier (AID) to the transponder whose answer it either positive or negative. In our case the select request is sent to the SEC, which distributes the request to all smartcard chips. If there is a positive responses the SEC then will redirect the communication flow accordingly. If there is no positive response, the SEC will return a negative answer to the reader. In case there are more than one positive response sent to the SEC, the SEC would forward a negative response to the reader and additionally contact the trusted service manager (TSM). In this case there is a management issue with the secure elements [7]. The SEC keeps a temporary list of AIDs and the appropriate secure elements in a registry, which is cleared in case the phone is rebooted. This helps the SEC to save time as the multi cast has to be performed only once.

When an external smartcard reader communicates with the NFC device, the SEC identifies the type of the reader by analyzing the commands/APDUs after the selection process has been completed and forwards the data stream to the appropriate smartcard chips. In case there is more than one instance of a smartcard type available, the SEC will consult the AID registry to route the data correctly. Thus the external reader is not able to distinguish between a smartcard and an NFC device.

An explicit selection of the appropriate secure element by the reader by indicating a communication through a logical channel is another option. This implementation would require, that there is fixed mapping between secure elements and logical channel IDs. Additionally, the reader has to know on which secure element there is the application the reader is looking for. This is the major issue with the mentioned implementation. Therefore logical channels are not further considered for explicit selection.

4.2 Internal Authentication of Secure Elements

Besides only having a single UID for contactless communication, the SEC also poses another advantage. The SEC can be used as a central instance in the device to perform the management of the secure elements. As the SEC connected through the SWP with the UICC, the SEC is able to make use of the wireless communication capabilities of the handset. This is an important feature with regard to the authorization of secure elements. The issue with a removable secure element, like the ones used in the Benq T80, is that the secure element can be inserted into any other mobile phone. As the authenticity of the phone is not validated by the secure element and vice versa, the vital feature of remote management of the secure elements is lost. The HCI is designed for a direct communication between different secure elements. This option allows a bilateral authentication between secure elements and/or the SEC. The UICC of the mobile phone serves as a gateway in order to retrieve certificates for the validation if necessary. By a simple example we demonstrate how the implemented system works on a JavaCard platform.

Each secure element, no matter if it is a UICC or a removable secure element such as a SD Card, contains an activation applet in the issuer security domain. This applet holds a public/private key pair and a certificate from the issuer and the appropriate root certificate for validation. Also the SEC contains a public/private key pair as well as a certificate from the issuer.

During the boot sequence of the mobile phone, the SEC first selects the activation applet on the UICC. Then a secure channel is established and a bilateral authentication is performed. In case this is the first time for the UICC communicating with the SEC, the UICC established a data connection to the CA/TSM in order to validate the certificate of the SEC. The public key to validate the certificate is kept by the UICC, hence the validation of the SEC does not require an OTA connection the next time. The validation of the UICC's certificate for the SEC is more difficult, as the SEC can not yet trust the UICC. However, the SEC instructs the UICC to establish a connection to the appropriate CA/TSM to validate the UICC's certificate. Then the SEC encrypts the UICC's certificate with its public key, signs it and adds an identifier. The external party processes the request, checks the certificate, encrypts the appropriate public key with the SEC's public key and also signs it. The SEC can verify the authenticity of the data by validating the signature of the package and additionally only the SEC is able to decrypt the data. The retrieved public key allows the SEC to validate the certificate of the UICC and sets the activation flag in the activation applet.

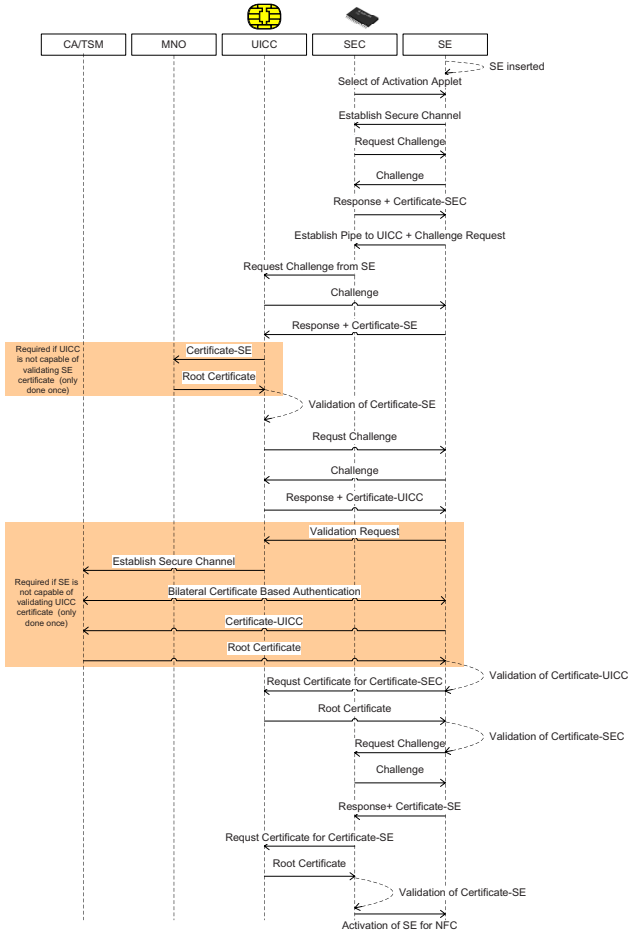


Fig. 4. Activation of an additional SE by the SEC for contactless communication

authenticate the other components again. With this implementation, the use of stolen SEs is prevented. The SE can store more than one certificate, allowing the user to change the SE between two handsets. Besides protecting the use of secure elements without permission, the platform manager (TSM) is always able to remotely access the secure elements OTA (Fig. 4).

5 Conclusion

The proposed a Secure Element Controller (SEC) enhances the compatibility of upcoming NFC devices with existing infrastructures. The implementation results in a single ID for NFC devices with multiple operating modes and multiple secure elements. The SEC takes over the routing of the data streams to the

appropriate secure element chip. The next step of our conception approach is the implementation of a SEC using the NFCBox [5].

There are several major benefits of our proposal. The most important one is the interoperability with already existing infrastructures. Although NFC is claimed to be compatible anyway, this is not the case when it comes to readers supporting only one card in the field. One of the most popular services using such a kind of reader is *MasterCard's PayPass*.

Secondly the integration of a SEC allows the parallel use of the peer-to-peer mode and the internal communication with the secure elements (internal mode; Fig.: 3(b)). In order to secure the plain NFC peer-to-peer data stream the use of a secure element to perform authentication or encryption is a reasonable feature, but not supported by the current architecture. Services that only rely on the fixed identifier of the smartcard are not feasible with this implementation, as the ID is partly random. However, systems only using the UID of contactless smartcards are unsecured anyway. This is due to the reason that the UID is primarily necessary for the anti-collision and selection of the transponder. Hence, no authentication or encryption is needed to read this ID.

Additionally the SEC can facilitate the authorization of secure elements as well as the OTA management. Hence, Trusted Service Managers are able to handle applications in a secure element other than the UICC and the he proposed authentication mechanisms avoids abuse of contactless applications. To sum up, the integration of a SEC would bring several benefits to an NFC device with regard to interoperability, security and manageability.

References

1. International Organization for Standardization: Near Field Communication - Interface and Protocol (NFCIP-1). ISO/IEC 18092 (2004)
2. EMVCo LLC: EMV Contactless Specifications for Payment Systems. PayPass ISO/IEC 14443 Implementation Specification (2006), <http://www.emvco.com/>
3. Kunkat, H.: NFC und seine Pluspunkte. *Electronic Wireless* 1, 4-8 (2005)
4. ETSI: Smart Cards: UICC-CLF interface; host Controller Interface. Draft (2007) (Release 7), www.etsi.org
5. Dillinger, O., Langer, J., Madlmayr, G., Muehlberger, A.: Near field communication in embedded systems. In: *Proceedings of the Embedded World Conference*, vol. 1, p. 7 (2006)
6. Bishwajit, C., Juha, R.: Mobile Device Security Element. Mobey Forum, Satamradankatu 3 B, 3rd floor 00020 Nordea, Helsinki/Finland (2005)
7. GSMA London Office 1st Floor, Mid City Place, 71 High Holborn, London WC1V 6EA, United Kingdom: mobile NFC technical guidelines. 2.0 edn, 1st Revision (2007)
8. Dillinger, O., Madlmayr, G., Schaffer, C., Langer, J.: An approach to nfc's mode switch. In: Dreiseitl, S., Hendorfer, G., Jodlbauer, H., Kastner, J., Mueller-Wipperferth, T. (eds.) *Proceedings of the Science Day of the University of Applied Sciences Upper Austria, FH OÖ F & E GmbH*, vol. 2, p. 6. Shaker Verlag, Aachen (2006)
9. Dawidowsky, F.: Method for operating a near field communication system. European Patent Office, EP 1 653 632 A1 (2006)

10. Rowse, G., Pendleburyrel, J.: Electronic near field communication enabled multifunctional device and method of its operation. Patent, US Patent Application Publication (2006)
11. Nohl, K.: Cryptanalysis of crypto-1 (2008), <http://www.cs.virginia.edu/~kn5f/Mifare.Cryptanalysis.htm>
12. International Organization for Standardization: Proximity cards. ISO/IEC 14443 (2003)

Coupon Recalculation for the GPS Authentication Scheme*

Georg Hofferek and Johannes Wolkerstorfer

Graz University of Technology,
Institute for Applied Information Processing
and Communications (IAIK),
Inffeldgasse 16a, 8010 Graz, Austria
Georg.Hofferek@iaik.tugraz.at,
Johannes.Wolkerstorfer@iaik.tugraz.at

Abstract. Equipping branded goods with RFID tags is an effective measure to fight the growing black market of counterfeit products. Asymmetric cryptography is the technology of choice to achieve strong authentication but suffers from its ample demand of area and power resources. The GPS authentication scheme showed that a coupon-based approach can cope with the limited resources of passive RFID tags. This article extends the idea of coupons by recalculating coupons during the idle time of tags when they are powered but do not actively communicate. This approach relaxes latency requirements and allows to implement GPS hardware using only 800 gate equivalents plus storage for 560 bytes. In the average case it has the same performance as the classical coupon-based approach but does not suffer its susceptibility to denial-of-service attacks.

1 Introduction

Radio frequency identification (RFID) is an emerging technology for optimizing logistic processes. Goods or pallets can be equipped with small and cheap RFID labels to give them an electronic identity. RFID labels can be read over air interfaces without direct line of sight. The main components of an RFID label are an antenna and an RFID tag. The minimalistic tag is a small chip containing an analog front-end which is connected to the antenna and a digital part. The chip receives its power for operation over the same antenna which is used for communication. Thus, the available power budget is very small. The functionality of tags is very often limited to basic operations like sending a unique ID upon request. More sophisticated tags may contain sensors or memories for storing more information about the product they are attached to.

A promising field of application for RFID tags is the authentication of goods to prove their genuineness. Providing unique IDs is a first step but does not

* The results presented in this article origin from the European Union funded FP7 project *Bridge* (IST-2005-033546).

solve the problem because the wireless air interface, which operates either on 13.56 MHz (HF) or around 900 MHz (UHF), can be easily eavesdropped. Thus, an attacker could obtain the UID of an original product and produce an infinite number of cloned tags having the same identity. Cryptographic authentication will inhibit such counterfeiting of goods. This is a fast growing market because the economical damage of counterfeit products exceeds \$600 billion annually [1].

During the last years many schemes have been proposed to bring strong cryptographic authentication to RFID tags. These activities focused on efficient hardware implementation because RFID tags have fierce constraints regarding silicon area and power consumption. The first choice was implementing symmetric cryptography, which has small footprint. A major work by Feldhofer et al. demonstrated the feasibility of implementing AES on passively powered RFID tags [4]. They achieved an AES encryption with a circuit complexity of 3500 gate equivalents taking 1000 clock cycles. On 0.35 μm CMOS technology, the encryption draws an average power of only 5 μW , which is well below the requirements of HF tags.

Although symmetric cryptography can be implemented efficiently, it suffers from the key distribution problem. In order to authenticate a product, the verifier must know the secret key. Thus, symmetric cryptography is only applicable in closed systems but not in world-wide logistic processes where not all involved parties are known in advance and in particular not all of them are trusted. Asymmetric cryptography overcomes many shortcomings of symmetric cryptography but requires a lot more resources. Area requirements are at least three times higher and the computation time in terms of clock cycles is usually more than hundred times higher. Most research on asymmetric cryptography for RFID centered on elliptic-curve cryptography, which offers reasonable security for RFID systems with key lengths of 113 to 256 bits [16,19].

The GPS authentication scheme, named after its authors Girault, Poupard, and Stern, offers the possibility to fill the gap between symmetric and asymmetric cryptography [5,6]. GPS is a public-key-based zero-knowledge protocol which is similar to Schnorr's scheme but uses a composite RSA-like modulus n . GPS is standardized in ISO/IEC 9798-5 [10]. There are also variants of GPS that are based on elliptic-curve cryptography. One of the remarkable properties of the GPS scheme is the possibility to use coupons. Coupons are lists of values which are needed during authentication. These values do not depend on any input from the interrogator, so they can be precomputed. That shifts the major computational load from the time of protocol execution to e.g. production time of the tag. §2 will go into the details of the GPS protocol. At this point it is only of interest that resource-constrained RFID tags can compute GPS authentications by having the ability to store a few coupons (each roughly between 200–500 bits for reasonable parameters) and to compute (non-modular) integer operations (addition, multiplication) with operand sizes of 160–260 bits.

The coupon approach can be efficiently implemented in hardware [12,13]. The promising results of [12,13] should not conceal profound drawbacks of the approach. First, only a limited number of authentications is possible because

storage is costly. Precomputed coupons have to be stored in non-volatile memory at the time of personalization. RFID tags usually use EEPROM memory as non-volatile memory. Memory sizes are kept at minimum to keep the cost of the tags low. Thus, only a few to a dozen of coupons are reasonable. This opens the door for denial-of-service attacks. Any RFID interrogator can request a GPS authentication from a tag, and few requests are sufficient to exhaust all coupons. This type of attack is difficult to prevent because the tag has no notion of time to limit the number of requests per time interval, nor does it have the possibility to authenticate interrogators. In order to authenticate an interrogator the tag would need the same computational resources as the reader. This would nullify the advantages of the coupon approach.

The coupon-recalculation approach presented in this article addresses these drawbacks. When RFID tags have the possibility to compute fresh GPS coupons during their idle time, an unlimited number of authentications is possible. Moreover the storage requirements can be lowered to store just a few coupons. When the comprehensive number-theoretic computations are done in the idle time, and thus well ahead of the actual use of the result, the requirements for the hardware change completely: Latency of the computation is no longer a key issue. Furthermore the processing can take comparatively long because RFID tags are usually long in the (electro-)magnetic field of the reader in comparison to the actual interrogation time. This allows to rethink hardware architectures for multi-precision modular arithmetic radically. We will show that tiny multiply-accumulate structures are sufficient to compute strong public-key cryptography on RFID tags.

The remainder of the article details the GPS authentication scheme and the coupon-recalculation approach in §2. §3 sketches the proposed hardware architectures. §4 presents algorithmic details and architectural details of a digit-serial approach optimized for small footprint and low power consumption. §5 presents achieved results and §6 concludes the article.

2 GPS Authentication

2.1 Basic Algorithm and Parameters

GPS authentication is a zero-knowledge authentication protocol that allows small hardware implementations of the prover wanting to assure its identity. A thorough analysis of the GPS authentication scheme, including comparisons with similar schemes is given in [9]. There, mathematical properties and background information are given, along with security considerations and references to almost all other literature about GPS. This section will only present the most important facts. The GPS protocol is depicted by Fig. 1. The prover first computes a commitment x , which is sent to the verifier. After having received the verifier's challenge c , the prover calculates the response y , which can then be checked by the verifier.

The parameters ρ , δ , and σ determine the bit lengths of the random value r , the challenge c , and the secret key s , respectively. Common values are $\sigma = 160$,

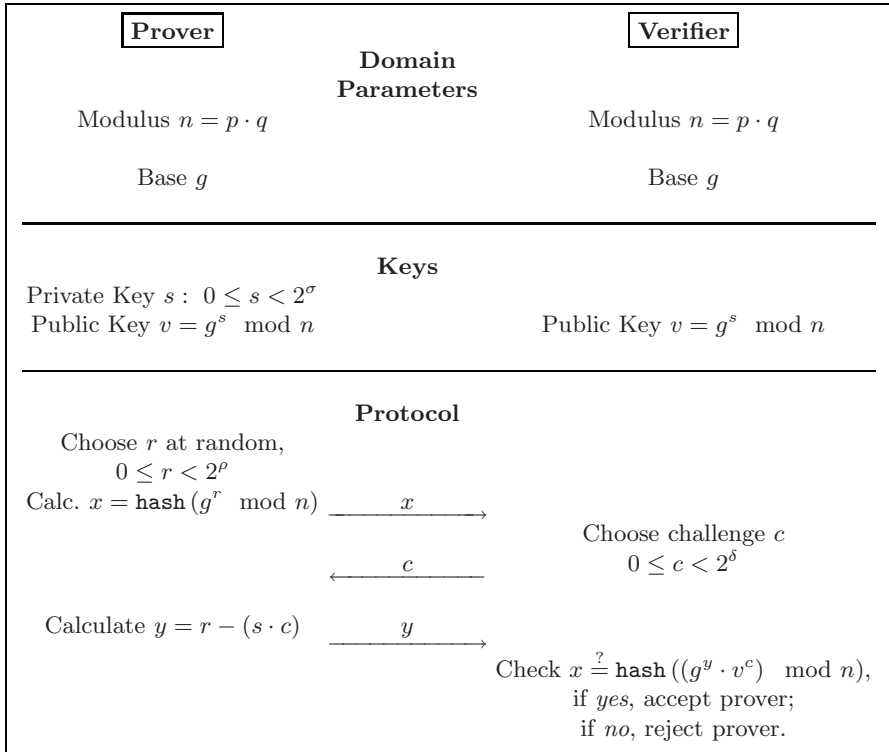


Fig. 1. Sketch of the basic GPS protocol, as standardized in [10]

$\delta = 20$, and $\rho = \sigma + \delta + 80 = 260$. The composite modulus n should be at least 1024 bits long. The integer subtraction in the last step can be substituted by addition if the public key is changed to $v = g^{-s} \pmod n$ (cf. [15]). This simplifies the computation and allows an implementation of the prover without requiring signed numbers. The hash function shown in Fig. 1 is optional [10]. It mainly reduces bandwidth requirements for transmitting the commitment. In ultra-low footprint implementations it can be omitted.

2.2 Coupon Approach

The computation of the commitment x does not depend on any input from the verifier. This computation can already be done ahead of the actual protocol execution. This observation was made by Girault in [7]. This paves the way for the so-called *coupon approach*. A set of *coupons* (r_i, x_i) can be precomputed (e.g. at production time of the tag) and stored in non-volatile memory. When the protocol interaction starts, the prover selects a coupon (r_i, x_i) from memory and sends x_i to the verifier. After receiving the challenge c , the prover can compute the response $y = r_i + s \cdot c$. McLoone et al. use this approach in their implementations [12,13]. They managed to implement the final integer operations

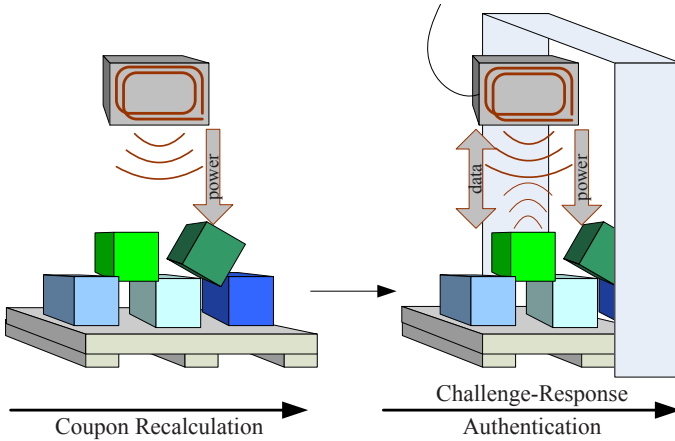


Fig. 2. Application scenario for the coupon recalculation approach

on a circuit size smaller than 1000 gate equivalents in less than 150 clock cycles by exploiting a special variant of GPS based on challenges with a low Hamming weight (cf. [8]).

However, these implementations are susceptible to denial-of-service attacks. Once all coupons are used, the tag cannot authenticate itself any more. Due to the authentication being only unilateral a tag cannot determine whether an authentication request from a reader is warrantable or not. If a tag contained k coupons, an attacker could disable the tag by requesting k authentications. Giving tags the ability to (re-)calculate coupons prevents such denial-of-service attacks.

2.3 GPS Coupon Recalculation Approach for RFID

The coupon recalculation approach is an extension of the coupon approach. Whenever a tag is idle, but still supplied with power, it can use the time to compute new coupons to refill its coupon storage. RFID tags are powered rather long in comparison to the actual data transmission times. Tags are in the electromagnetic field of readers for seconds, while the protocol interaction completes within milliseconds. This also applies for high volume logistic processes like the one shown in Fig. 2.

In situations where it is very likely that all coupons have been used and thus need to be recalculated, RFID tags can be supplied with power before interrogation (see Fig. 2). Power supply requires only the transmission of the carrier frequency (e.g. 13.56 MHz in HF systems). No sophisticated reader circuitry is necessary to drive the powering antennas. A simple oscillator is sufficient as interrogator circuit because neither modulation nor demodulation is needed.

The application scenario of coupon recalculation changes the requirements for the hardware radically. In contrast to previous asymmetric cryptographic

hardware neither latency nor throughput are of particular interest. Instead, hardware implementations can focus on minimizing silicon area and power consumption. The footprint of the circuit is of particular interest because the circuit size has also an linear impact on the power consumption. Thus, the smallest possible hardware suits the RFID requirements best. In order to keep the computation time within limits, the metric $A \cdot t \cdot P$, which weighs area, time, and power equally, was used to find an optimum hardware architecture.

Besides minimizing the area of the circuit it is also interesting to consider the maximum clock frequency f_{max} . Although RFID tags clock their digital circuitry at low clock frequencies to keep the power consumption low, high f_{max} can be used to accelerate coupon recalculation when a higher power budget is available. This is the case when a single good has to be authenticated. In such a situation the reader antenna is usually held next to the RFID tag. The power density of electro-magnetic fields are at least four times higher when halving the distance between the tag and the antenna. The digital circuit of RFID tags is almost ever optimized for worst-case situations at the far end of the reader field. Detecting higher power densities and exploiting them by increasing the clock frequency gives an RFID tag with GPS hardware the possibility to improve the average performance. Anyhow, next we will analyze hardware approaches assuming constant clock frequency.

3 Hardware Architecture

A hardware implementation of the recalculation approach can focus on the power efficiency and on low footprint optimization. For computing a fresh coupon (r, x) , a modular exponentiation $x = g^r \bmod n$ has to be computed. The square-and-multiply algorithm is the algorithm of choice for hardware implementations, which computes $x = \prod_{i=0}^{\log_2 r} g^{ir_i}$.

3.1 Full-Precision Architecture

An approach often used for (high-speed) hardware is to use a bit-serial full-precision multiplication to compute the modular multiplications and squarings. Many implementations even use digit-serial approaches to improve latency [3]. Most hardware implementations use Montgomery multiplication [14] instead of normal integer multiplication and subsequent modular reduction. Montgomery multiplication simplifies modular reduction. Bit-serial multiplication schedules one operand at full precision, while the other one is processed bit by bit. The disadvantage of this approach is that at least five full-precision registers are necessary. Three registers are needed for the multiplication: two for the input values, one for the (intermediate) result; one register is needed to store an auxiliary variable which is necessary for implementing the square-and-multiply algorithm, and one register is needed to store the modulus. Assuming a modulus of 1024 bits, this would mean that a full-precision architecture would need at least 5120

flip-flops, or more than 30 000 gate equivalents. After adding the necessary components for partial-product generation and accumulation, the complete datapath of a full-precision arithmetic unit requires approximately 50 000 gate equivalents (not counting non-volatile memory for storing domain parameters). When sticking to full-precision architectures, there are no more significant improvements to be made concerning the circuit's size. The size is mainly determined by the storage requirements, which depend on the size of the modulus. It is thus reasonable to concentrate on digit-level arithmetic architectures that operate on smaller word sizes, which will be discussed in the next section.

3.2 Digit-Level Architecture

Bit-serial or digit-serial hardware architectures strive for improving performance. In the recalculation approach, performance is of secondary interest. This allows to use hardware architectures with smaller footprint and lower power consumption. Multiply-accumulate structures known from signal processing and instruction-set extensions allow to implement multiple-precision arithmetic at minimal hardware costs.

Fig. 3 shows the architecture of a GPS-enhanced RFID tag that uses a digit-level arithmetic unit to implement multiply-and-accumulate operations. There are two obvious reasons why a digit-level approach decreases the circuit's size: First the arithmetic unit itself is much smaller because its data width is only that of one digit (8 to 64 bits), and not of full precision. Second, the operands and temporary results can be stored in a hard-macro RAM, which is more area efficient than flip-flop-based storage. Most hard-macro RAM circuits are offered either as single-port or dual-port memories. The little area expense of the second port leads to great time savings in our application. Thus, a dual-ported RAM is used. One port reads a digit as input for the arithmetic unit while the other port is used to write back computed results at a different address.

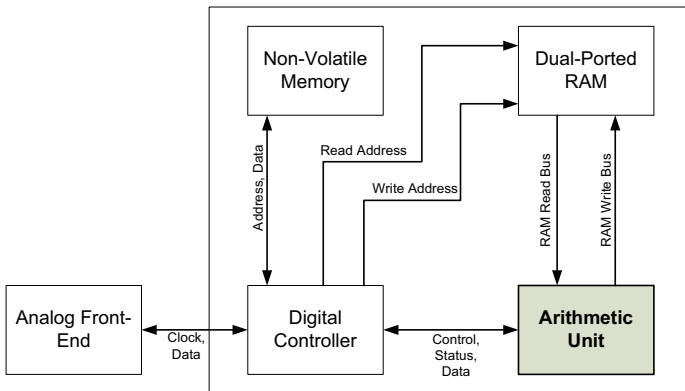


Fig. 3. Overview of an RFID tag, enhanced with a digit-level GPS architecture

Non-volatile memory is also necessary, to store domain parameters (e.g. secret key, modulus) and coupons. A *digital controller* implements the necessary algorithms to perform the overall computation, by means of digit-level operations which are carried out by the *arithmetic unit*. The arithmetic unit contains the actual datapath for performing the required operations. §4 details the arithmetic unit. The width of the data buses shown in Fig. 3 is equal to the digit size of the arithmetic unit. Thus one digit can be loaded from and stored to the RAM in every cycle. The digit size is a parameter fixed at synthesis time. Reasonable sizes are between 8 and 64 bits. Larger digit sizes seem impractical because the corresponding $d \times d$ -digit multiplier would be too large. Digit sizes below 8 bits also seem impractical since that would cause too long computations. Computation time scales quadratically $\mathcal{O}(k^2)$ with the number of digits $k = \frac{n}{d}$ to represent a multi-precision word.

4 Arithmetic Unit

The GPS arithmetic unit must be capable of performing modular exponentiation ($x = g^r \bmod n$) for the coupon recalculation, and (non-modular integer) addition and multiplication ($y = r + s \cdot c$) for the response calculation during protocol execution. Our approach computes modular exponentiation in a sequence of square-and-multiply operations, which are broken down to digit level. The implementation efficiency of (modular) multiplication is crucial for the GPS hardware. It determines the circuit size and its performance.

4.1 Digit-Level Montgomery Multiplication

In our approach modular multiplication is implemented by Montgomery multiplication. There are several ways to break down Montgomery multiplication to digit-level operations. Five different approaches have been analyzed by Koç et al. in [11]. They differ in how much primitive operations (digit additions, digit multiplications, memory read/write) and how much temporary memory they require. The one which seems most suitable for our work, due to its low memory requirements and low number of operations, is referred to by Koç et al. as *Coarsely integrated operand scanning* (CIOS). [11] explains its details very well. Basically the CIOS algorithm works as follows: The first digit of the first operand is multiplied with all digits of the second operand to obtain a partial product. After that an interleaved reduction step takes place by adding a multiple of the modulus to the partial product to make it divisible by 2^d , and subsequently shifting the intermediate result one digit to the right (= division by 2^d). Then the next accumulation step is executed: The next partial product (obtained by multiplying the second digit of the first operand with all digits of the second operand) is added to the intermediate result. This is followed by another interleaved reduction step. These accumulation and reduction steps are repeated for all remaining digits of the first operand.

4.2 Analyzing Digit-Level Operations

To perform the desired operations the arithmetic unit consists of two (or more) *input registers* (of size d) for the operands, and two *output registers* (of size d). Two output registers are necessary because digit-level operations can produce results which are larger than the digit size; two output registers are also sufficient, since no atomic digit-level operation of the CIOS algorithm from [11] can produce results larger than $2^{2d} - 1$. No extra carry registers are needed. The dominant operation performed during one GPS protocol execution is modular exponentiation, which is broken down into multiplications (by means of square-and-multiply). Single multiplications are performed using the CIOS algorithm. The dominant operation in this algorithm lies within the accumulation step. This operation calculates the sum of three terms: The current value of the **Output High** register, one digit from RAM, and the product of two more digits from RAM; i. e. the result is $D_1 + D_2 \cdot D_3 + H$, where D_i denote digits from RAM and H denotes the value of the **Output High** register at the time before the operation starts. All other (digit-level) operations (within the CIOS algorithm and within algorithms for multi-precision addition and multiplication) can be reduced to this operation. E.g. calculating just the product of two digits can be achieved by setting D_1 and H to 0. Our proposed arithmetic unit is designed to compute the operation $D_1 + D_2 \cdot D_3 + H$ most efficiently.

4.3 Schematic

The operation described above has three external inputs D_1, D_2, D_3 , thus the most obvious architecture for an arithmetic unit has three input registers. For such a unit it would be possible to wire the arithmetic components (multiplier, adder) in a way that the operation could be executed within one single cycle. That is of course assuming that the input registers already hold the values D_1, D_2, D_3 . Since only one digit per cycle can be loaded from a RAM hard-macro, spending the area for three input registers does not bring real speed-up. Thus it is reasonable to use only two input registers.

The schematic of the arithmetic unit is depicted in Fig. 4. A high-level model written in Java proved the efficiency of the approach. An implementation in Verilog is used for synthesis. Both the high-level model and the Verilog model are parameterizable concerning the digit size d . The $d \times d$ -digit multiplier is implemented as pure combinational logic; its area demand scales quadratically with the digit size d . However, since only small digit sizes will be considered, this approach seems possible. Other multiplier architectures (e.g. bit-serial multiplication) would require additional resources (e.g. shift registers). They would also complicate control and prolong execution time. It should be noted that the implementation details of the multiplier have been left to the synthesis tool. A simple `assign c = a * b` statement was put into the Verilog code, so that the synthesis tool would have the possibility to optimize the circuit while considering area and clock constraints. The input register D_2 has additional logic to be able to load either a value coming from the RAM read-bus (`din`), or one of

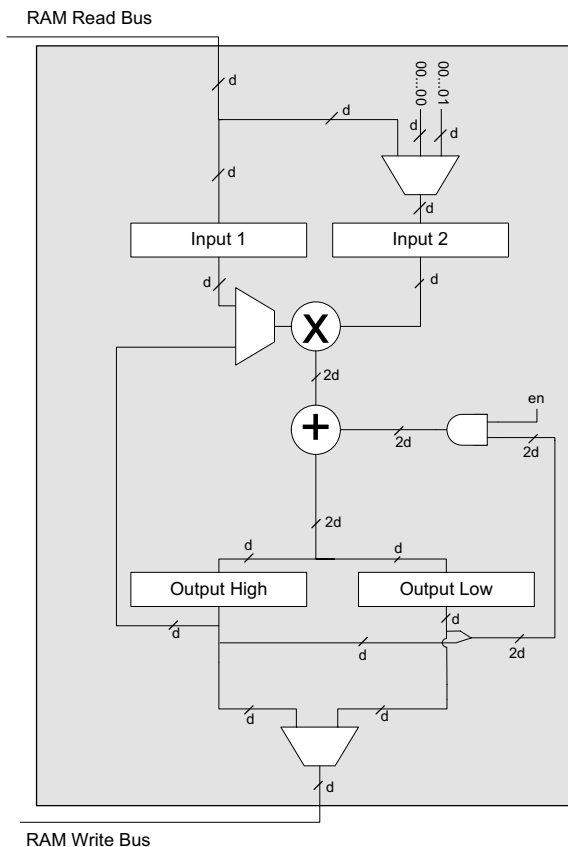


Fig. 4. Schematic of the arithmetic unit with digit size d

the two constant values $(000 \dots 00)_2 = 0$ and $(000 \dots 01)_2 = 1$. The reason for this is explained in the next section. Although Fig. 4 shows a multiplexer, the most efficient implementation which achieves this behavior consists of d AND gates and one OR gate. Another possibility would be to store the constants $(000 \dots 00)_2 = 0$ and $(000 \dots 01)_2 = 1$ in memory and load them from there. This approach does not need additional logic in the arithmetic unit but the minimum size of the memory is increased by two entries. Furthermore the execution time of the overall computation will be increased with this approach because loading constant values into the second input register allocates the read bus of the memory. Meanwhile, no meaningful operation of the datapath is possible.

Let us now investigate how the arithmetic unit depicted in Fig. 4 can be used to compute the modular exponentiation $x = g^r \bmod n$. As it has been explained, exponentiation is broken down to modular multiplication by the square-and-multiply approach. These multiplications are in turn broken down to digit-level operations. The dominant digit-level operation is $D_1 + D_2 \cdot D_3 + H$, as described

Table 1. Comparison of synthesis results of the arithmetic unit (not including RAM) for different CMOS technologies, and two different digit sizes

Digit Size: 32 bits
(RAM for 151 digits needed, approx. 4.7 million clock cycles necessary for computing one coupon)

Technology	Area [μm^2]	Critical Path [ns]
AMS 0.35 [2]	361 561.20	28.93
UMC 0.25 [18]	181 470.96	20.94
UMC 0.13 [17]	44 115.45	10.86

Digit Size: 8 bits
(RAM for 560 digits needed, approx. 66.6 million clock cycles necessary for computing one coupon)

Technology	Area [μm^2]	Critical Path [ns]
AMS 0.35	43 880.47	10.16
UMC 0.25	20 599.92	5.98
UMC 0.13	4 855.68	3.41

in §4.2, where D_i denote digits in RAM and H denotes the current value of the Output High Register. The sum $D_1 + H$ can be accumulated in two steps by means of the feedback loops shown in Fig. 4. While doing so, the Input 1 register is loaded with the constant value $(000\dots01)_2 = 1$. That way the multiplier output is 1 times its left input. Then in the third step the values D_2, D_3 are loaded to the input registers and the product $D_2 \cdot D_3$ is added to the intermediate result $D_1 + H$, thus resulting in the final result $D_1 + D_2 \cdot D_3 + H$. All other digit-level operations are performed in a similar way.

5 Results

Our implementation of an arithmetic unit for coupon recalculation of the GPS authentication scheme is parameterizable with respect to the digit size d . Smaller digit sizes lead to smaller arithmetic units, but the time necessary to complete one authentication increases heavily with smaller digit sizes. The number of necessary digit-level operations is roughly proportional to the square of the number of digits per operand. Therefore in our opinion digit sizes below 8 bit are unpractical, considering that the full-precision size of the operands is (at least) 1024 bits.

Table 1 shows the synthesis results for the implementation of the arithmetic unit. For synthesis the *PKS Shell* from Cadence was used. Three different CMOS technologies have been used: A 0.35 μm CMOS technology from austriamicrosystems [2], and a 0.25 μm and a 0.13 μm CMOS technology from UMC [17][18]. The critical path of the circuit is very short, due to the small arithmetic components, which are only of digit size. That means that it could be clocked very fast; frequencies up to 290 MHz are possible with UMC 0.13 μm CMOS technology. This opens a wide window of possibilities for operation. When operated at very low clock frequencies (e.g. 100 kHz), the circuit consumes very little power (6.25 μW). We have performed a power simulation of the arithmetic unit (digit size: 8 bit), using the near-SPICE simulator *Nanosim* from Synopsys. The circuit was first synthesized for AMS 0.35 μm CMOS technology with *PKS Shell*, then placed and routed with *First Encounter*. The resulting layout can be seen

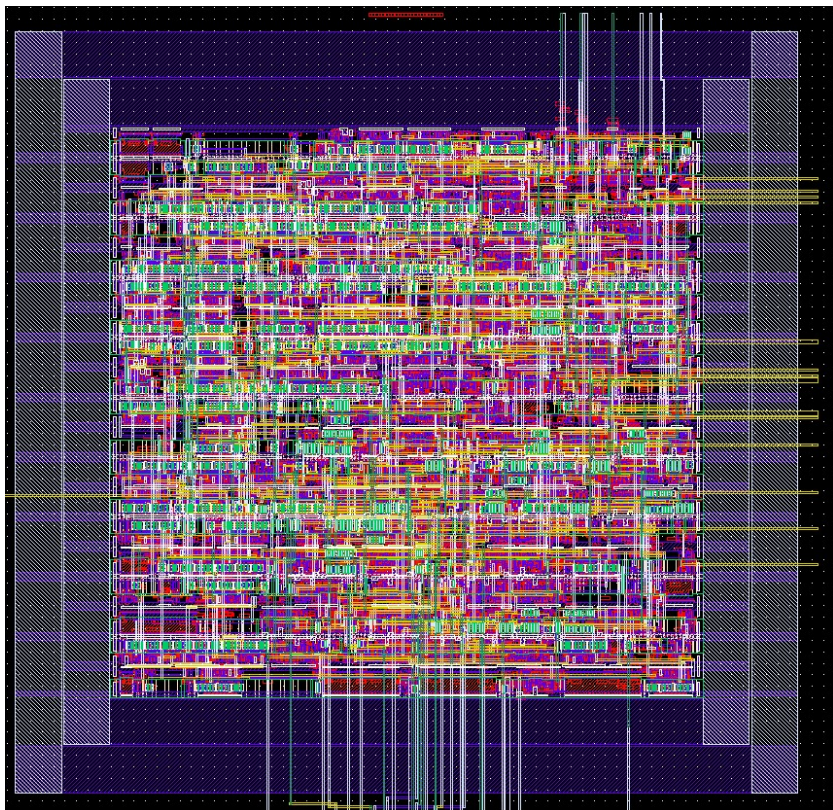


Fig. 5. Layout of the GPS arithmetic unit (digit size: 8 bits) after place-and-route

in figure 5. A netlist for *Nanosim* was extracted with *Assura* from the layout after place-and-route. The simulation reveals that the arithmetic unit consumes approximately $2.5 \mu\text{A}$ when clocked with 100 kHz. Thus, with 2.5 V supply voltage, the power consumption is $6.25 \mu\text{W}$. This is comparable to the results of Feldhofer et al. for AES [4]. They use the same CMOS technology and their AES implementation consumes $5 \mu\text{W}$.

When raising the supply voltage to 3.3 V the performance of the circuit improves. It can be clocked with frequencies up to 125 MHz. At 125 MHz and 3.3 V supply voltage, the circuit draws a current of 3.2 mA. That means that if more power is available the computation can be sped up by a factor of 1250, compared to operation at 100 kHz.

The synthesis of the 8-bit arithmetic unit requires approximately 800 gate equivalents. After place-and-route the circuit takes up an area of approximately $64250 \mu\text{m}^2$. In addition, it would require a RAM hard-macro capable of storing 560 bytes. In comparison, the smallest implementation by McLoone et al. [12,13] requires only 431 gate equivalents. However their implementation can only compute the response $y = r + s \cdot c$ in the last step of the GPS protocol, whereas our

circuit is capable of calculating the modular exponentiation $x = g^r \bmod n$ to compute new coupons on-tag. Of course our implementation can also be used to calculate the response y . When using standard sizes for the parameters (cf. §2.1) this calculation takes 627 cycles, including time to load the operands to RAM. 33 of these cycles can be saved if a fresh coupon has been computed immediately before and thus the random value r is already present in RAM.

6 Conclusion

This paper introduced the coupon recalculation approach, which is an area- and power-efficient way to extend the *coupon approach* of the GPS authentication scheme. During the idle time of RFID tags, fresh coupons are computed for future authentication requests. A full-precision arithmetic unit would require approximately 50 000 gate equivalents, plus approximately 2200 bits of non-volatile memory to store keys and domain parameters (if typical values for GPS parameters are used). Our digit-level approach makes use of an arithmetic unit, which requires only approximately 800 GE for a digit size of 8 bits. In addition RAM resources for storing 560 bytes are needed. One coupon calculation takes about 66.6 million clock cycles, and the maximum clock frequency of the circuit on UMC 0.13 μm CMOS technology is approximately 290 MHz. The approach is very flexible and can be used either when very little power is available (i.e. low clock frequency, longer execution time). When more power is available, the computation can be accelerated by increasing the clock frequency. Up to four coupons can be computed per second under such conditions (290 MHz).

References

1. ADT Tyco Fire & Security, Alien Technology, Impinj Inc., Intel Corporation, Symbol Technologies, and Xterprise. RFID and UHF – A Prescription for RFID Success in the Pharmaceutical Industry. White paper (2006)
2. Austriamicrosystems. 0.35 μm CMOS Process Standard-Cell Library, http://asic.austriamicrosystems.com/databooks/index_c35.html
3. Blum, T., Paar, C.: Montgomery modular exponentiation on reconfigurable hardware. In: Koren, Kornerup (eds.) Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, pp. 70–77. IEEE Computer Society Press, Los Alamitos (1999)
4. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEEE Proceedings on Information Security 152(1), 13–20 (2005)
5. Girault, M.: An identity-based identification scheme based on discrete logarithms modulo a composite number. In: Daamgard, I. (ed.) EUROCRYPT 1990. LNCS, vol. 473, pp. 481–486. Springer, Heidelberg (1991)
6. Girault, M.: Self-certified public keys. In: Davies, D. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 490–497. Springer, Heidelberg (1991)
7. Girault, M.: Low-size coupons for low-cost ic cards. In Smart Card Research and Advanced Applications. In: Proceedings of the Fourth Working Conference on Smart Card Research and Advanced Applications, CARDIS 2000, Bristol, UK, September 20–22, 2000, vol. 180, pp. 39–50. Kluwer, Dordrecht (2000)

8. Girault, M., Lefranc, D.: Public key authentication with one (online) single addition. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 413–427. Springer, Heidelberg (2004)
9. Girault, M., Poupard, G., Stern, J.: On the fly authentication and signature schemes based on groups of unknown order. *Journal of Cryptology* 19(4), 463–487 (2006)
10. ISO/IEC. International Standard ISO/IEC 9798 Part 5: Mechanisms using zero-knowledge techniques (December 2004)
11. Koç, C.K., Acar, T., Kaliski, B.J.: Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*. 16(3), 26–33 (1996)
12. McLoone, M., Robshaw, M.: Public key cryptography and RFID tags. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377. Springer, Heidelberg (2006)
13. McLoone, M., Robshaw, M.J.B.: New architectures for low-cost public key cryptography on RFID tags. In: IEEE International Symposium on Circuits and Systems (ISCAS 2007), pp. 1827–1830 (May 2007)
14. Montgomery, P.L.: Modular multiplication without trial division. *Mathematics of Computation* 44, 519–521 (1985)
15. NESSIE. Final report of European project number IST-1999-12324, named new european schemes for signatures, integrity, and encryption (April 2004), <https://www.cosic.esat.kuleuven.be/nessie/Bookv015.pdf>
16. Tuyls, P., Batina, L.: RFID-Tags for Anti-counterfeiting. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 115–131. Springer, Heidelberg (2006)
17. UMC. UMC standard cell library — 130 nm CMOS process
18. UMC. UMC standard cell library — 250 nm CMOS process
19. Wolkerstorfer, J.: Is Elliptic-Curve Cryptography Suitable for Small Devices? In: Workshop on RFID and Lightweight Crypto, Graz, Austria, July 13–15, 2005, pp. 78–91 (2005)

Provably Secure Grouping-Proofs for RFID Tags^{*}

Mike Burmester¹, Breno de Medeiros², and Rossana Motta¹

¹ Florida State University, Tallahassee FL 32306, USA

{burmester, motta}@cs.fsu.edu

² Google Inc., 1600 Amphitheatre Pkwy Mountain View, CA 94043

breno@google.com

Abstract. We investigate an application of RFIDs referred to in the literature as *group scanning*, in which several tags are “simultaneously” scanned by a reader device. Our goal is to study the group scanning problem in strong adversarial models. We present a security model for this application and give a formal description of the attending security requirements, focusing on the privacy (anonymity) of the grouped tags, and/ or forward-security properties. Our model is based on the Universal Composability framework and supports re-usability (through modularity of security guarantees). We introduce novel protocols that realize the security models, focusing on efficient solutions based on off-the-shelf components, such as highly optimized pseudo-random function designs that require fewer than 2000 Gate-Equivalents.

1 Introduction and Previous Work

Radio Frequency Identification (RFID) tags were initially developed as small electronic hardware components whose main function is to broadcast a unique identifying number upon request. The simplest type of RFID tags are *passive* devices—i.e., without an internal power source of their own, relying on an antenna coil to capture RF power broadcast by an RFID reader. In this paper, we focus on tags that additionally feature a basic integrated circuit and memory. This IC can be used to process challenges issued by the RFID reader and to generate an appropriate response. For details on these tags, and more generally on the standards for RFID systems, the reader is referred to the Electronic Protocol Code [10] and the ISO 18000 standard [11].

The low cost and high convenience value of RFID tags gives them the potential for massive deployment. Accordingly, they have found increased adoption in manufacturing (assembly-line oversight), supply chain management, inventory control, business automation applications, and in counterfeit prevention. Initial designs of RFID identification protocols focused on performance issues with lesser attention paid to resilience and security. As the technology has matured and found application into high-security and/or high-integrity settings, the need for support of stronger security features has been recognized. Many works have looked into the issue of secure identification and authentication, including [1, 2, 5, 6, 7, 8, 9, 13, 14, 15, 19, 20, 21, 22].

^{*} Part of this material is based on work supported by the U.S. Army Research Laboratory, and the U.S. Research Office under grant number DAAD 19-02-1-0235.

Ari Juels introduced the security context of a new RFID application—which he called a yoking-proof [12], that involves generating evidence of simultaneous presence of two tags in the range of an RFID reader. As noted in [12], interesting security engineering challenges arise in regards to yoking-proofs when the trusted server (or Verifier) is not online during the scan activity. The first proposed protocol introduced in [12] was later found to be insecure [4, 18]. Yoking-proofs have been extended to *grouping-proofs* in which groups of tags prove simultaneous presence in the range of an RFID reader—see e.g. [4, 17, 18]. In this paper, we examine the latter solutions and identify similar weaknesses in their design.

Our main contribution in this paper is to present a comprehensive security framework for RFID grouping-proofs, including a formal description of the attending security requirements. In previous work, the group scanning application has only been described at relatively informal levels, making it difficult to provide side-to-side comparisons between alternative proposals. We then construct practical solutions guided by the security requirements and constraints of this novel model.

As Juels already pointed out, there are several practical scenarios where grouping-proofs could substantially expand the capabilities of RFID-based systems. For example, some products may need to be shipped together in groups and one may want to monitor their progress through the supply chain—e.g., of hardware components or kits. Other situations include environments that require a high level of security, such as airports. In this case, it may be necessary to couple an identifier, such as an electronic passport, with a physical person or with any of his/her belongings, such as their bags. In battlefield contexts, weaponry or equipment may have to be linked to specific personnel, so that it may only be used or operated by the intended users.

In some of the above scenarios, the RFID reader may not enjoy continuous connectivity with the trusted Verifier, and delayed confirmation may be acceptable. For instance, this may be the case with supply chain applications, due to the increased fragmentation and outsourcing of manufacturing functions. A supplier of partially assembled kits may perform scanning activities that will be verified later when the kits are assembled at a different site. Therefore, efficient and optimized realizations of this primitive that achieve strong security guarantees—such as we describe in this paper—are practically relevant contributions in the design space of RFID protocols.

2 RFID Deployments and Threat Model

A typical deployment of an RFID system involves three types of legitimate entities: *tags*, *readers* and a *Verifier*. The tags are attached to, or embedded in, objects to be identified. In this paper we focus on passive RFID tags that have no power of their own but have a small footprint CMOS integrated circuit, ROM, RAM and non-volatile EEPROM. The RFID readers typically contain a *transceiver*, a *control unit* and a *coupling element*, to interrogate tags. They implement a radio interface to the tags and a high level interface to the Verifier that processes captured data.

The Verifier (a back-end server) is a trusted entity that maintains a database containing the information needed to identify tags, including their identification numbers. In our protocols, since the integrity of the whole RFID system is entirely dependent on the

proper behavior of the Verifier, we assume that the Verifier is physically secure and not attackable.

Grouping-proofs involve several tags being scanned by an RFID reader in the same session. The reader establishes a communication channel that links the tags of a group and enables the tags to generate a proof of “simultaneous presence” within its broadcast range. The proof should be verifiable by the Verifier. Throughout this paper, we assume the following about the environment characterizing group scanning applications:

- *The tags are passive*, i.e., have no power of their own, and have very limited computation and communication capabilities. However, we assume that they are able to perform basic cryptographic operations such as generating pseudo-random numbers and evaluating pseudo-random functions.
- *RFID tags do not maintain clocks or keep time*. However, the activity time span of a tag during a single session can be limited using techniques such as measuring the discharge rate of capacitors, as described in [12].
- *RFID readers establish communication channels* that link the tags of a group. This takes place at the data link layer of the RFID network: after tags that claim to belong to a group are “identified” (tags may use pseudonyms) a common (wireless) channel linking the tags via the reader is established.
- *RFID readers are trusted to manage the interrogation of tags*. They enable the tags of a group to generate a grouping proof during an interrogation session, and keep a record of such proofs for each session. These records cannot be manipulated by the adversary. In the offline case readers must also store private information regarding interrogation challenges obtained from the Verifier.
- *The Verifier is a trusted entity*, that may share some secret information with the tags such as cryptographic keys. The Verifier has a secure channel (private and authenticated) that links it to the (authenticated) RFID readers.
- *Grouping proofs are only valid if they are generated according to their protocol in the presence of an authorized RFID reader*. In particular if the flows of the protocol are ordered, the ordering cannot be violated. Also, proofs generated during different sessions are not valid (even if correct).

The Verifier can be online or offline and different solutions are required in each case. We further distinguish between online *fully-interactive* mode and online *batch* mode. In fully-interactive mode the Verifier can receive and send messages to specific tags throughout the protocol execution. In contrast, the interaction of the Verifier in batch mode is restricted to broadcasting a challenge that is valid for a (short) time span, collecting responses from the tags (via RFID reader intermediates), and checking for legitimate group interactions—the Verifier in batch mode never unicasts messages to particular groups of tags.

It is straightforward to design solutions for the fully-interactive mode of the grouping-proof problem—indeed, it is sufficient for individual tags to authenticate themselves to the Verifier, which will then decide on the success of the grouping-proof by using auxiliary data, e.g., the tag identifiers of the groups. Therefore, research on grouping-proofs has focused on the offline case, with some results also targeted at the online batch modality. Accordingly, in this paper, we focus on offline solutions, except for the forward-secure protocol, where we only describe a solution in the online batch mode.

2.1 Attacks on RFID Tags

Several types of attacks against RFID systems have been described in the literature. While each of these are types known in other platforms, unique aspects of the RFID domain make it worthwhile to discuss them anew.

- *Denial-of-Service* (DoS) attacks: The adversary causes tags to assume a state from which they can no longer function properly.
- *Unauthorized tag cloning*: The adversary captures keys or other tag data that allow for impersonation.
- *Unauthorized tracing*: The adversary should not be able to trace and/or recognize tags.
- *Replay attacks*: The adversary uses a tag's response to a reader's challenge to impersonate the tag.
- *Interleaving and reflection attacks*: These are *concurrency* attacks in which the adversary combines flows from different instantiations to get a new valid transcript.

These attacks are exacerbated by the mobility of the tags, allowing them to be manipulated at a distance by covert readers.

2.2 The Threat Model for RFID

The extremely limited computational capabilities of RFID tags imply that traditional multi-party computation techniques for securing communication protocols are not feasible, and that instead lightweight approaches must be considered. Yet the robustness and security requirements for RFID applications can be quite significant. Ultimately, security solutions for RFID applications must take as rigorous a view of security as other types of applications. Accordingly, our threat model assumes a Byzantine adversary. In this model all legitimate entities (tags, readers, the Verifier) and the adversary have polynomially bounded resources. The adversary controls the delivery schedule of the communication channels, and may eavesdrop into, or modify, their contents, and also instantiate new channels and directly interact with honest parties.

We are mainly concerned with security issues at the protocol layer and not with physical or link layer issues—For details on physical/link layer issues the reader is referred to [10, 11].

2.3 Guidelines for Secure RFID Applications

Below we present effective strategies that can be used to thwart the attacks described in Section 2.1. These strategies are incorporated in the design of our protocols.

- *DoS attacks*: One way to prevent the adversary from causing tags to assume an unsafe state is by having each tag share with the Verifier a permanent secret key k_{tag} , which the tag uses to generate a response when challenged by an RFID reader.

- *Cloning attacks*: The Verifier should be able to check a tag’s response, but the adversary should not be able to access a tag’s identifying data. This can be assured by using cryptographic one-way functions.
- *Unauthorized tracing*: The adversary should not be able to link tag responses to particular tags. This can be guaranteed by (pseudo-)randomizing the values of the tags’ responses.
- *Interleaving and Replay attacks*: The adversary should not be able to construct valid transcripts by combining flows from different sessions. This can be assured by binding all messages in a session to the secret key and to fresh (pseudo-)random values.
- *Generic concurrency-based attacks*: Protocols that are secure in isolation may become vulnerable under concurrent execution (with other instances of itself or of other protocols). To guarantee security against such attacks it is necessary to model security in a concurrency-aware model. In this paper, we use the Universal Composability model.

3 Previous Work: RFID Grouping-Proofs

In this section we describe three grouping-proofs proposed in the literature and discuss their vulnerabilities.

3.1 The Yoking-proof

This is a proof of simultaneous presence of two tags in the range of a reader [12]. The reader scans the tags sequentially. The tags have secret keys known to the Verifier but not the reader, and counters, and use a keyed message authentication code and a keyed hash function to compute a “*yoking-proof*”. Saito and Sakurai observed [18] that a minimalist version of this proof (that does not use counters) is subject to an interleaving attack. The attack was shown [4] to extend to the full version of the proof, but it was also shown that it can be easily be prevented.

There are two other weaknesses we shall discuss here. The first concerns the fact that the tags do not (and cannot) check each other’s computation. This implies that in the offline mode unrelated tags can participate in a yoking session, and that the failure will only be detected by the Verifier at some later time, not by the reader. While, from an authentication perspective, this may not represent a security threat, in many practical applications it is an undesirable waste of resources, and could be characterized as a DoS vulnerability. To appreciate how accidental pairing may create challenges to real-world applications—e.g., where yoking is used to ensure that components are grouped in a shipment, consider the following scenario. A reader is configured to take temporary measures after a failed yoking attempt, e.g., notify an assembly worker of a missing component in a shipment pallet. This capability is denied if a tag (either accidentally or maliciously) engages in yoking sessions with unrelated tags, and possibly even with itself—for the latter, we refer the reader to the modified re-play attack scenario described in [18]. Accidental occurrences of this type might not be unlikely, in particular with anonymous yoking-proofs, and they are facilitated by the fact that the

scanning range of readers may vary according to different environmental conditions. In order to prevent this kind of vulnerability, in our protocols we use a group secret key k_{group} , which is shared by all the tags belonging to that group¹.

A more serious weakness concerns the nature of the “proof” P_{AB} generated by the tags: this is *not* a proof that tag_A and tag_B were scanned simultaneously while in the presence of an authorized reader. Indeed, one cannot exclude the possibility that P_{AB} was generated while the tags were in the presence of a rogue reader, and that at a later time P_{AB} was replayed by a corrupted tag (impersonating successively tag_A and tag_B) in the presence of the authorized reader. To avoid this kind of attack in our protocols the challenge of authorized tags will include a nonce (r_{sys}).

3.2 Proofs for Multiple RFID Tags

These extend yoking-proofs to handle arbitrary number of tags in a group [18] and use time-stamps, to thwart re-play attacks. Piramuthu [17] replaced the time-stamps by random numbers. This is important, because time-stamps can be predicted, allowing for attacks that collect prior responses and combine them to forge proofs of simultaneous interaction. As with the yoking-proofs, these fail to satisfy the security guidelines in Section 2.3. In particular, the random numbers used are vulnerable to a multi-proof session attacks [16].

3.3 Clumping-proofs for Multiple RFID Tags

[16]. These combine the strengths of yoking-proofs and multiple tag proofs and address some of their weaknesses. The tags use counters and the reader uses a keyed hash of a time-stamp, obtained from the Verifier, to make its requests unpredictable. For details, we refer the reader to [16].

Clumping-proofs use counters to reduce the search complexity of the Verifier. However their value is updated regardless of the received flows, so they can be incremented arbitrarily by the adversary (via rogue readers). Therefore, they cannot be relied upon to identify tags, and in the worst case an exhaustive search through the keys may have to be used. A security proof is provided in the Random Oracle Model [3]. However, this does not address concurrency threats, a substantial limitation of the analysis, considering that the original yoking-proofs [12] admit a similar security proof and are vulnerable to concurrency-based attacks.

4 Our Protocols: Robust Grouping-Proofs

We present three RFID grouping proofs. The first one does not provide anonymity, the second adds support to anonymity and the third improves on the second by incorporating forward-secrecy.

¹ Although group keys will prevent faulty tags from participating in a grouping-proof that involves non-faulty tags, they cannot prevent *malicious* tags from submitting an invalid proof to a reader, since proofs can only be verified by the Verifier. Our last protocol (Section 4.3), in which the groups are authenticated by the reader, addresses this issue.

In the first protocol, the proof sent from the tags to the reader and from the reader to the Verifier includes a group identifier ID_{group} . For the second protocol, no identifier is passed to the reader: the proof uses values that depend on the group’s identifier and key and on the Verifier’s challenge but the dependency is known only to the Verifier. Thus, only the Verifier is able to match the proof with a given group of tags: this guarantees unlinkability and anonymity. In the third protocol the secret keys and the group keys of the tags are updated after each execution, thus providing forward-secrecy.

There are two reasons why we present different protocols. First, prior work on group scanning has considered both the anonymous and non-anonymous settings. Since anonymizing protocols requires additional computational steps and correspondingly larger tag circuitry, simpler alternatives are preferred whenever anonymity is not a concern. Second, the introduction of protocols of increasing complexity follows a natural progress that facilitates the understanding of the protocols structure.

Although for simplicity we illustrate our protocols with two tags, the extension to any number of tags is straightforward. Irrespective of the number of tags involved, a specific tag in the group always plays the role of “initiator,” transmitting either a counter (in the non-anonymous protocol), a random number, or a random password (in the other versions). This has the security benefit of curtailing reflection attacks. To implement this feature, it is not necessary that tags engage in any sort of real-time agreement protocol, it is sufficient to hard-code the behavior of tags.

We consider situations in which the Verifier is not online while the tags are scanned. Each tag stores in non-volatile memory two secret keys (both shared with the Verifier): a *group key* k_{group} used to prove membership in a group, and an *identification key* k_{tag} used to authenticate protocol flows. Tags instances are denoted as tag_A or tag_B , and the key for instance tag_A is written in shorthand as k_A .

Each protocol starts with a reader broadcasting a random challenge r_{sys} , which is obtained from the trusted Verifier at regular intervals. This challenge defines the scanning period, i.e., each group should be scanned at most once between consecutive challenge values. In other words, the Verifier cannot (without further assumptions) determine simultaneity of a group scan to a finer time interval than the scanning period.

4.1 A Robust Grouping-Proof

Our first non-anonymous grouping-proof is presented for two tags, tag_A and tag_B , where tag_A is the initiator tag—see Fig. 1. The current state of the group is determined by a counter c stored by the initiator tag. The counter is updated with each execution of the protocol. Each group is assigned an identifier ID_{group} and the Verifier stores these values together with the private keys of each tag in a database $D = \{(ID_{tag}, k_{tag}, k_{group})\}$. The protocol has three phases. In the first phase the reader challenges the tags in its range with r_{sys} and the tags respond with their group identifier ID_{AB} . In the second phase—which takes place at the data-link layer—the tags are linked by channels through the reader. In the third, the tags prove membership in their group.

Each phase can be executed concurrently with all the tags in the group, except that the third phase must be initiated by the initiator tag (tag_A in the diagram). The various phases cannot be consolidated without loss of some security feature. If we remove the first phase (r_{sys}) the protocol would be subject to a “full-replay” attack (Section 2.3).

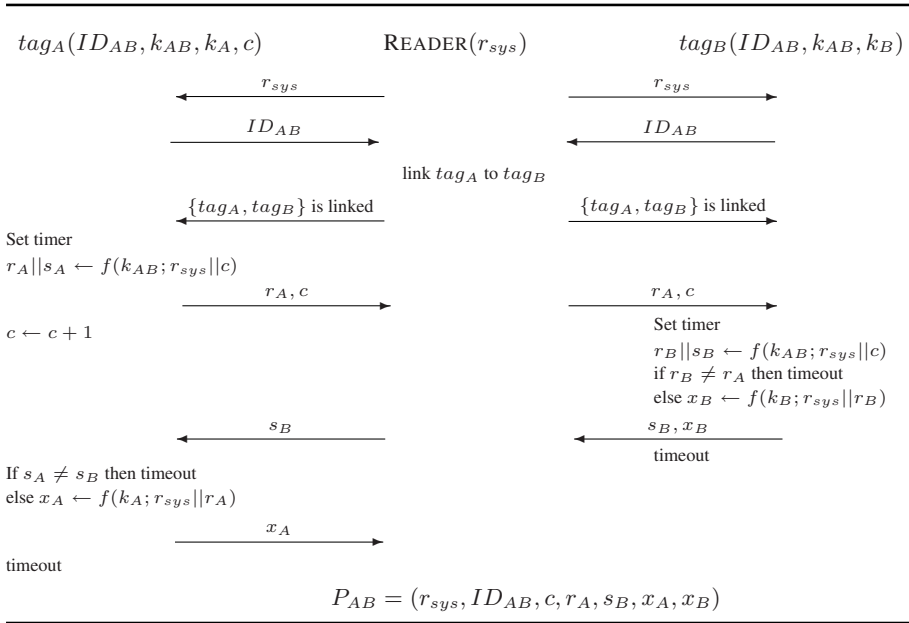


Fig. 1. A robust grouping-proof—for two tags

If we remove the second phase (the exchange of ID_{AB}), the reader would be unable to match the tags. Phase three consists of three rounds of communication, and each is crucial to provide the data for the proof. If we were to suppress the exchange of s_B and x_B , or if we did not implement the timeout, then replay attacks would be successful. Also, the implementation of the third round enables an authorized reader to detect certain protocol failures immediately, namely those that lead the initiator tag to timeout. The update of the counter c immediately after it is sent by tag_A allows the state to be updated even if the protocol round should be interrupted. This, along with timers prevents replay attacks.

The extension of the protocol to more than two tags is achieved as follows. In the first and second phases, the reader communicates with all tags concurrently. In the first round of the third phase, the reader communicates only with the initiator tag; it communicates with all other tags concurrently in the second round; and again with the initiator tag in the third round, providing it with concatenated answers from the second round.

In the protocol each tag_X uses its group key k_{AB} to evaluate $f(k_{AB}; r_{sys} || c)$, where f is a pseudo-random function and “||” denotes concatenation. This is parsed to get numbers r_X, s_X of equal length, used to identify the parties of the group and prove membership in the group. Tags use their secret key to confirm correctness of the proof. The proof of simultaneous scanning is $P_{AB} = (r_{sys}, ID_{AB}, c, r_A, s_B, x_A, x_B)$. In our protocol, it is possible for an authorized reader to know whether grouped tags were actually scanned or not because, in the latter case, one or more of the tags would timeout. This represents an improvement over the past protocols, in which the success or failure of the yoking- or grouping-proof, could only be detected by the Verifier. This

protocol can be implemented very efficiently, with a footprint of fewer than 2000 Gate-Equivalents. For a discussion on optimized implementations of pseudo-random functions suitable for RFID applications, we refer the reader to [13].

Security Analysis. The universal composability (UC) framework defines the security of a protocol in terms of its simulatability by an idealized functionality \mathcal{F} (which can be thought of as specifications of the achievable security goals for the protocol). \mathcal{F} is a trusted entity that can be invoked by using appropriate calls. We say that a protocol ρ UC-realizes \mathcal{F} , if for any adversary \mathcal{A} , any real-world simulation of ρ in the presence of \mathcal{A} can be emulated by an ideal-world simulation in which the adversary invokes \mathcal{F} , in such a way that no polynomial-time environment \mathcal{Z} can distinguish between the two simulations. In ideal-world simulations, the adversary has access to all the outputs of \mathcal{F} , as in the real-world it can eavesdrop into all communications.

For our first protocol the functionality \mathcal{F}_{group} comprises the behavior expected of a grouping-proof. It is invoked by five calls: **activate**, **initiate**, **link**, **complete**, and **verify**. The first call is used by the environment \mathcal{Z} to activate the system by instantiating the Verifier, an authorized reader and some tags. Note that keys initially shared between the Verifier and the tags are not under control of the adversary in this model—in the UC model this is called a trusted setup. The second call is used by readers to initiate an interrogation session, and corresponds to an r_{sys} challenge, and by tags to declare their group membership. The call **link**, links the tags specified in **activate**, and the call **initiate** for tags gives their response to the reader’s challenge. The call **complete** is for initiator tags and completes a proof: it corresponds to x_A . The call **verify** can be used to submit a putative proof transcript to the Verifier. The adversary can arbitrarily invoke \mathcal{F}_{group} and mediates between all parties involved in its interactions with \mathcal{F}_{group} .

All the outputs resulting from calls to \mathcal{F}_{group} , except for the tag calls that produce identifiers, are random strings. The functionality keeps a record of every output string, and uses these strings in the same way as the protocol ρ uses the corresponding outputs. \mathcal{F}_{group} will only accept (verify) those proofs that it has generated itself during a particular session as a result of the activation of the system, the initiation and linking by an authorized reader, the initiation of all the tags that belong to a particular group, and the completion by an initiator tag (in this order). In the full version of this paper we shall show that our first protocol UC-realizes the \mathcal{F}_{group} functionality.

4.2 A Robust Anonymous Grouping-Proof

For our second protocol, group identifiers are replaced by randomized group pseudonyms ps_{group} . To protect against de-synchronization failure or attacks, one (or more in groups of $n > 2$ tags) of the tags must maintain both a current and an earlier version of the state of their pseudonyms. For this purpose all tags in a *group* store in non-volatile memory one or more values of a pseudo-random number r_{tag} .² Initiator tags store only the current value, while the other tags store two values, r_{tag}^{old} , r_{tag}^{cur} . These

² We use r_{tag} instead of r_{group} to distinguish between the actions of individual tags in *group* during the execution of the protocol. The values of these numbers are the same for all tags in *group* when the adversary is passive.

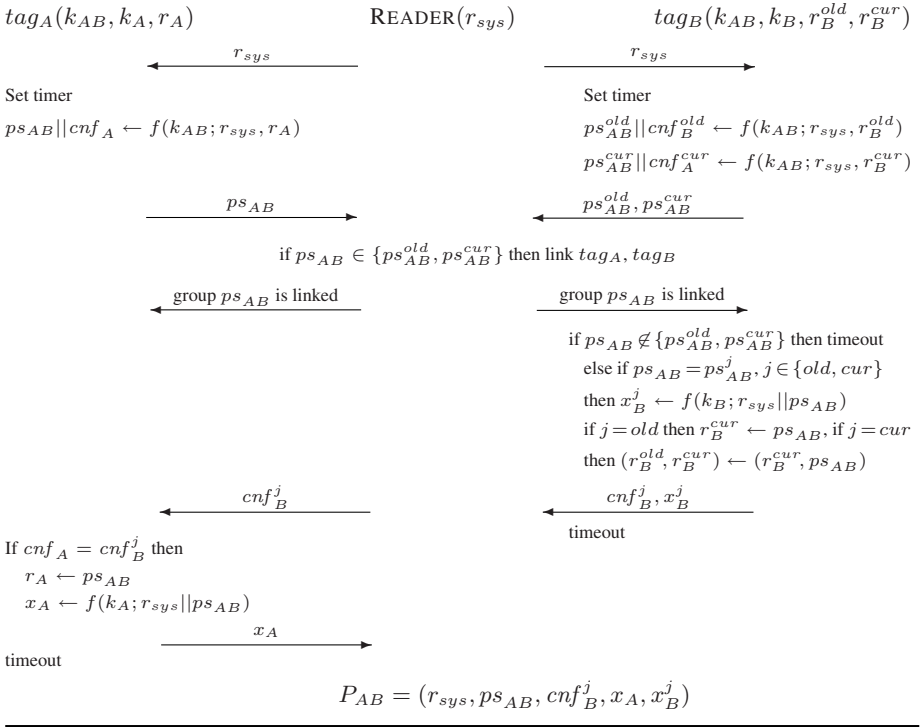


Fig. 2. A robust anonymous grouping-proof—for two tags

values are used to compute the group pseudonym. First $f(k_{group}; r_{sys} || r_{tag})$ is evaluated, where r_{sys} is the random challenge of the Verifier. Then, this is parsed to get two numbers ps_{group}, cnf_{tag} , of equal length, where cnf_{tag} is a confirmation used to authenticate the pseudonym. Initiator tags compute one pseudonym ps_{group} ; the other tags compute two pseudonyms ps_{group}^{old} and ps_{group}^{cur} (in a similar way).

The tags in *group* update the value(s) of their group pseudonym(s) with each successful execution of their part of the grouping protocol. The protocol is presented in Fig. 2, where tag_A is the initiator and for simplicity we depict only one additional tag, tag_B . It is easy to see how this protocol can be extended to groups of $n > 2$ tags. In particular, the reader will link all the tags for which at least one pseudonym is ps_{group} , provided there are n such tags.

The Verifier keeps a database $D = (r_{sys}, \{(k_{tag}, k_{group}, ps_{group})\})$ that links, for session r_{sys} , the secret key of each tag to its group key and the group pseudonym of the corresponding initiator tag. The pseudonyms are updated with each successful execution of the protocol (using the next value of r_{sys}). The database D is also used to optimize the performance of the protocol: if the adversary has not challenged the tags of *group* since their last interaction (e.g., via rogue readers), then the value of the pseudonym in D will be the one that is actually used by the initiator tag, and therefore

the corresponding secret keys can be found directly (one lookup) and used to verify the correctness of the authenticator x_{tag} of the initiator tag. The secret keys of the other tags in $group$ can be found in the database D from the group key k_{group} , and used to verify the correctness of their authenticators. If no value in D corresponds to the pseudonym used by the initiator tag then the Verifier will have to find the secret key of the initiator from its authenticator $x_{tag} = f(k_{tag}; r_{sys} || ps_{group})$ by exhaustive search over all secret keys (of initiator tags). The pseudo-random numbers r_{tag} are initialized with random values r_A : for the initiator tag_A : $r_{tag} \leftarrow r_A$, while for all other tag_X in its group: $(r_X^{old}, r_X^{cur}) \leftarrow (r_A, r_A)$.

Observe that initiator tags respond with only one pseudonym and therefore can be distinguished from other tags (which respond with two pseudonyms). There are several ways to address this privacy issue, if it is of concern. One way is to assign to all tags a pair of pseudonyms, and identify groups by selecting those sets of tags that have one pseudonym ps^* in common. There will always be at least one tag in this set for which $ps_{group}^{cur} = ps^*$. The reader elects an initiator tag among those tags sharing the common pseudonym deterministically, probabilistically, or in some ad hoc way: e.g., the first to respond. The reader informs the initiator tag of its selection and indicates to the other tags which pseudonym ps^* is current. In this modification of the protocol all rounds are executed concurrently.

As in the previous protocol, each step is essential. The main difference is that in the anonymous protocol, the tags exchange pseudonyms ps_{AB} and $ps_{AB}^{old}, ps_{AB}^{cur}$, rather than a group identifier. The functionality provided by this step, however, is analogous in the two protocols and enables the Verifier to identify the group.

It is important to notice that even though the values that the reader receives for each completed round vary, if a malicious reader interrupts the session (round), preventing the pseudonym update, and then re-uses r_{sys} , it can link the two scannings. However, the power of this attack is limited because a single round with a non-faulty reader at any point will restore unlinkability. We shall refer to this property as, *session unlinkability*. More formally we have:

Definition 1. *An RFID protocol has session unlinkability if, any adversary, given any two tag interrogations int_1, int_2 , (not necessarily complete, or by authorized readers), where int_1 takes place before int_2 , and a history of earlier interrogations, cannot decide (with probability better than $0.5 + \varepsilon$, ε negligible) whether these involve the same tag or not, provided that either:*

- *The interrogation int_1 completed normally (successfully), or*
- *An interrogation of the tag involved in int_1 completed successfully after int_1 and before int_2 .*

Security Analysis. The functionality \mathcal{F}_{sa_group} of our second protocol comprises the behavior expected of an anonymous grouping-proof with session unlinkability. The functionality \mathcal{F}_{sa_group} is that same as \mathcal{F}_{group} except that:

³ A temporal relationship, as observed by the adversary. Note that if the adversary observes two interrogations overlapping in time, it can certainly assert that they do not belong to the same tag, since tags are currently technologically limited to single-threaded execution.

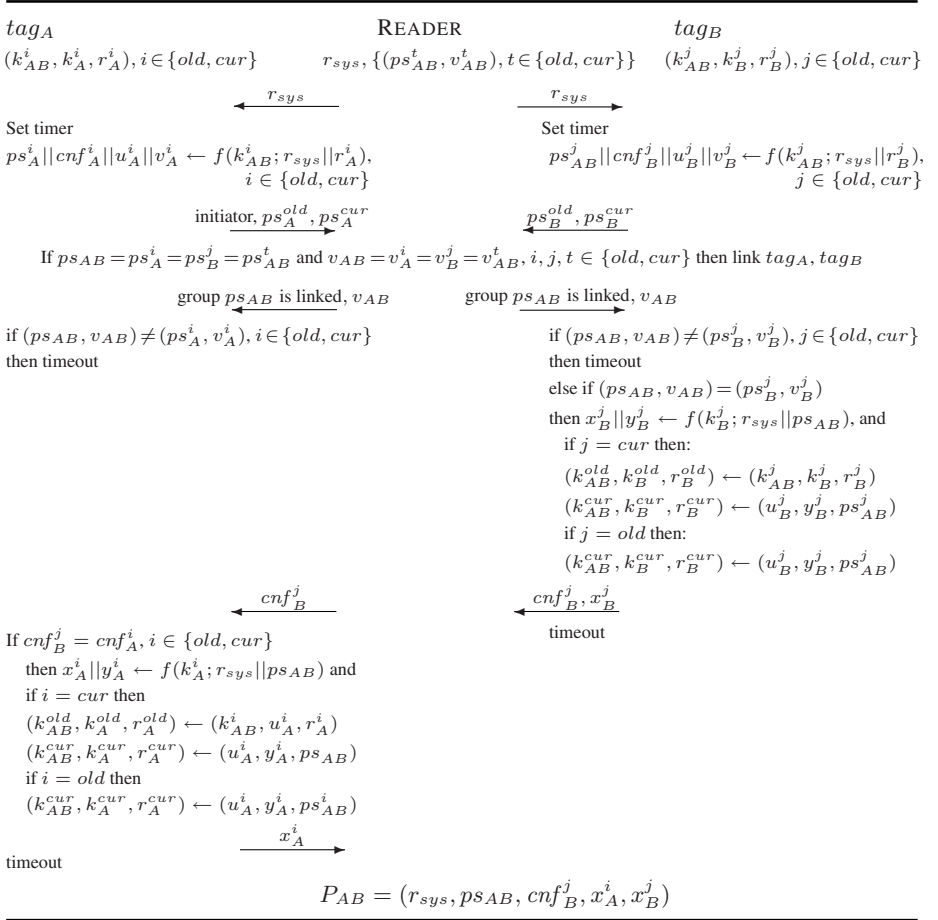


Fig. 3. An anonymous grouping-proof with forward-security—for two tags

1. The outputs of all its invocations are random numbers, including tag identifiers.
2. If a tag is initiated with the same reader challenge in a session, as in an earlier session that it was not allowed to complete (and no sessions with this tag completed in the interim), then \mathcal{F}_{sa_group} will output identical values.

This means that the adversary can link the (uncompleted) scannings throughout a given session. However in the next session, \mathcal{F}_{sa_group} will use a different (random) number, so linkability does not extend to any other sessions. In the full version of this paper we shall show that our second protocol UC-realizes the \mathcal{F}_{sa_group} functionality.

Notice that our second protocol is not able to provide forward-security: secrecy is no longer guaranteed if the secret keys are compromised.

4.3 A Robust Grouping-Proof with Forward-Secrecy

In our last protocol—see Fig. 3 the secret keys and the group keys of tags are updated after each protocol execution for forward-secrecy. All tags, including initiator tags, store two pairs of keys: group keys k_{group}^i and secret keys k_{tag}^i , $i \in \{old, cur\}$, as well as a pair of random numbers r_{tag}^i , $i \in \{old, cur\}$. The Verifier stores in a database D the current values $(r_{sys}, \{(k_{tag}^t, k_{group}^t, ps_{group}^t), t \in \{old, cur\}\})$: this allows it to link the values of the keys of each tag to the corresponding group pseudonym. At the end of each r_{sys} challenge session, the entries in D of all tags in the groups for which the reader has returned a valid proof P_{group} are updated: $(k_{tag}^t, k_{group}^t, ps_{group}^t) \leftarrow (y^t, u^t, r^t)$, $t \in \{old, cur\}$, using the equal-length parsings $f(k_{group}^t; r_{sys} || r_{tag}^t) = r^t || s^t || u^t || v^t$ and $f(k_{tag}^t; r_{sys} || r^t) = x^t || y^t$ (the use of the other parsed values is explained below). Since there are no non-volatile values to anchor the key and pseudonym updates to, we shall use the update chain itself as an anchor. This means that the state of the tags and the Verifier must be synchronized. In particular the adversary should not be able to manipulate valid group scans so as to de-synchronize the system. There are several ways in which this can be achieved. The solution we propose is to have the Verifier (a trusted party) give all authorized readers a table $\hat{D} = (r_{sys}, \{(ps_{group}^t, v_{group}^t), t \in \{old, cur\}\})$ whose values can be used to authenticate authorized readers to tags. The entries in this table are obtained by parsing $f(k_{group}^t; r_{sys}, r_{tag}^t) = r^t || s^t || u^t || v^t$ as above, and assigning: $(ps_{group}^t, v_{group}^t) \leftarrow (r^t, v^t)$, $t \in \{old, cur\}$. In this case however the *next* value of the challenge r_{sys} is used in the evaluation of f . The values in \hat{D} are updated for *all* groups in the system, at the beginning of each *new* r_{sys} session.

The protocol is given in Fig. 3 for two tags, tag_A, tag_B , with tag_A the initiator. In this protocol, adversarial readers cannot disable tags permanently by de-synchronizing them from the Verifier, because the tags discard old key values $k_{group}^{old}, k_{tag}^{old}, r_{tag}^{old}$ only after the Verifier has confirmed that it has updated its corresponding values. More specifically, if the reader is not adversarial, then $ps_{AB}^{cur} = ps_A^{cur} = ps_B^{cur}$, and the tags will update both current and old key and number values. If the reader is adversarial and has not returned the proof P_{AB} then $ps_{AB}^{cur} \neq ps_A^{cur}$, or ps_B^{cur} , and the updates will not affect old values, which therefore remain the same as those stored in the database \hat{D} . The state of the tags will only return to stable when an authorized reader returns a valid proof to the Verifier. Note that due to the state synchronization requirements, the protocol in Fig. 3 can only be implemented in online batch mode, not true offline mode. In the full paper, we discuss the batch offline case.

Forward-secrecy applies to periods during which the groups of tags are scanned by authorized readers that are not faulty. More specifically, a group of tags that is compromised can be traced back to the first interaction after the last non-faulty scanning session, and no further. We shall refer to this property as, *forward session-secrecy*. More formally we have:

Definition 2. *An RFID protocol has forward session-secrecy if session unlinkability holds for all sessions int_1 and int_2 as in Defn. 1 provided that either int_1 successfully completed prior to the corresponding tag(s) being compromised, or that a later session of the tag(s) involved in int_1 completed successfully prior to its (their) compromise.*

Security Analysis. The functionality \mathcal{F}_{fss_group} of our third protocol comprises the behavior expected of an anonymous grouping-proof with forward session-secrecy. The functionality \mathcal{F}_{fss_group} models key compromise, that is it allows for adaptive corruption. Otherwise it is similar to \mathcal{F}_{sa_group} . This means that the adversary can link incomplete scanings of non-compromised tags throughout a given session, but that this does not extend to other sessions. In the full version of this paper we shall show that our third protocol UC-realizes the \mathcal{F}_{fss_group} functionality.

5 Conclusion

Our main contribution in this paper is to present a security model for the group scanning problem. In previous work, this application has been described at relatively informal levels, making it difficult to provide side-to-side comparisons between alternative proposals. We have proposed three grouping-proofs that are provably secure in a very strong setting that guarantees security under concurrent executions and provides for safe re-use as a building block for more complex protocols. These proofs are also practically feasible, requiring only pseudo-random functions, which can be instantiated very efficiently in integrated circuits using a variety of primitives as a starting point, such as pseudo-random number generators or block ciphers.

References

1. Ateniese, G., Camenisch, J., de Medeiros, B.: Untraceable RFID tags via insubvertible encryption. In: Atluri, V., Meadows, C., Juels, A. (eds.) Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005, pp. 92–101. ACM, New York (2005)
2. Avoine, G., Oechslin, P.: A Scalable and Provably Secure Hash-Based RFID Protocol. In: 3rd IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2005 Workshops), pp. 110–114. IEEE Computer Society, Los Alamitos (2005)
3. Bellare, M., Rogaway, P.: Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In: ACM Conference on Computer and Communications Security, pp. 62–73 (1993)
4. Bolotnyy, L., Rose, G.: Generalized Yoking-Proofs for a group of Radio Frequency Identification Tags. In: International Conference on Mobile and Ubiquitous Systems, MOBIQUITOUS 2006, San Jose, CA (2006)
5. Bono, S.C., Green, M., Stubblefield, A., Juels, A., Rubin, A.D., Szydlo, M.: Security analysis of a cryptographically-enabled RFID device. In: Proc. USENIX Security Symposium (USENIX Security 2005), pp. 1–16. USENIX (2005)
6. Burmester, M., van Le, T., de Medeiros, B.: Provably secure ubiquitous systems: Universally composable RFID authentication protocols. In: Proceedings of the 2nd IEEE/CreateNet International Conference on Security and Privacy in Communication Networks (SECURECOMM 2006), IEEE Press, Los Alamitos (2006)
7. Dimitriou, T.: A lightweight RFID protocol to protect against traceability and cloning attacks. In: Proc. IEEE Intern. Conf. on Security and Privacy in Communication Networks (SECURECOMM 2005). IEEE Press, Los Alamitos (2005)
8. Dimitriou, T.: A secure and efficient RFID protocol that can make big brother obsolete. In: Proc. Intern. Conf. on Pervasive Computing and Communications (PerCom 2006). IEEE Press, Los Alamitos (2006)

9. Engberg, S.J., Harning, M.B., Jensen, C.D.: Zero-knowledge device authentication: Privacy & security enhanced rfid preserving business value and consumer convenience. In: Proceedings of Second Annual Conference on Privacy, Security and Trust (PST 2004), October 13-15, 2004, pp. 89–101. Wu Centre, University of New Brunswick, Fredericton (2004)
10. EPC Global. EPC tag data standards, vs. 1.3, http://www.epcglobalinc.org/standards/EPCglobal_Tag_Data_Standard_TDS_Version_1.3.pdf
11. ISO/IEC, <http://www.hightechnaid.com/standards/18000.htm>
12. Juels, A.: Yoking-Proofs for RFID tags. In: PERCOMW 2004: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops, pp. 138–142. IEEE Computer Society, Washington (2004)
13. Van Le, T., Burmester, M., de Medeiros, B.: Universally composable and forward-secure RFID authentication and authenticated key exchange. In: Bao, F., Miller, S. (eds.) Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2007), Singapore, March 20-22, 2007, pp. 242–252. ACM, New York (2007)
14. Molnar, D., Soppera, A., Wagner, D.: A scalable, delegatable pseudonym protocol enabling ownership transfer of RFID tags. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897. Springer, Heidelberg (2006)
15. Oren, Y., Shamir, A.: Power analysis of RFID tags. In: RSA Conference, Cryptographer's Track (RSA-CT 2006) (2006), <http://www.wisdom.weizmann.ac.il/~yossio/rfid>
16. Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda, A.: Solving the simultaneous scanning problem anonymously: clumping proofs for RFID tags. In: Third International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing, SecPerl 2007, Istanbul, Turkey, IEEE Computer Society Press, Los Alamitos (2007)
17. Piramuthu, S.: On existence proofs for multiple RFID tags. In: IEEE International Conference on Pervasive Services, Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing – SecPerU 2006, Lyon, France, June 2006. IEEE Computer Society Press, Los Alamitos (2006)
18. Saito, J., Sakurai, K.: Grouping Proof for RFID Tags. In: 19th International Conference on Advanced Information Networking and Applications (AINA 2005), Taipei, Taiwan, 28-30 March 2005, pp. 621–624. IEEE Computer Society, Los Alamitos (2005)
19. Sarma, S.E., Weis, S.A., Engels, D.W.: RFID systems and security and privacy implications. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002. LNCS, vol. 2523, pp. 454–469. Springer, Heidelberg (2003)
20. Tan, C.C., Sheng, B., Li, Q.: Severless Search and Authentication Protocols for RFID. In: Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2007), White Plains, New York, USA, 19-23 March 2007, pp. 3–12. IEEE Computer Society, Los Alamitos (2007)
21. Tsudik, G.: YA-TRAP: Yet Another Trivial RFID Authentication Protocol. In: 4th IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2006 Workshops), Pisa, Italy, 13-17 March 2006, pp. 640–643. IEEE Computer Society, Los Alamitos (2006)
22. Vajda, I., Buttyan, L.: Lightweight authentication protocols for low-cost RFID tags. In: Proc. Workshop on Security in Ubiquitous Computing (UBICOMP 2003) (2003)

Secure Implementation of the Stern Authentication and Signature Schemes for Low-Resource Devices

Pierre-Louis Cayrel¹, Philippe Gaborit¹, and Emmanuel Prouff²

¹ Université de Limoges, XLIM-DMI,
123, Av. Albert Thomas 87060 Limoges Cedex France
{pierre-louis.cayrel,philippe.gaborit}@xlim.fr

² Oberthur Technologies
71-73, rue des hautes pâtures 92726 Nanterre Cedex France
e.prouff@oberthurcs.com

Abstract. In this paper we describe the first implementation on smart-card of the code-based authentication protocol proposed by Stern at Crypto'93 and we give a securization of the scheme against side channel attacks. On the whole, this provides a secure implementation of a very practical authentication (and possibly signature) scheme which is mostly attractive for light-weight cryptography.

1 Introduction

While Cryptography aims at preventing persons from cheating, Coding Theory has been originally designed to prevent accidental errors coming from the imperfections of the transmission systems (*e.g.* phone lines, microwaves, satellite communications, CDs, etc.). Nowadays, it studies more generally how to protect information transiting over unperfect channels from alterations. The core idea is to send over the channel more data than the initial amount of information to convey. The added information, usually called *redundancy*, is structured in such a way that it is possible to detect and (eventually) to correct almost all the errors that could occur during the data transmission.

After a first scheme proposed by McEliece in 1978 using error-correcting codes for encryption, the idea of using error-correcting codes for authentication purposes was due to Harari, followed by Stern (first protocol) and Girault. The protocols of Harari and Girault were subsequently broken, while Stern's one was five-pass and unpractical. Eventually, the first practical and secure protocol based on error-correcting codes was proposed by Stern at Crypto'93 [15]. This zero-knowledge authentication protocol is based on an error-correcting codes problem usually referred as the *Syndrome Decoding* (*SD* in short) Problem. Stern's protocol is a Fiat-Shamir-like protocol but with a cheating probability of $2/3$ rather than $1/2$ for Fiat-Shamir. It is hence considered as a good alternative to the numerous authentication schemes whose security relies on classical number theory problems such as the factorization or the discrete logarithm problems.

Although the Stern Scheme was proposed almost 15 years ago, it has (as far as we know) never been implemented on smart card until now. This is merely due to the usual drawback of code-based systems: the size of the public data is large. Indeed, since the prover and the verifier have to know a large random matrix with at least 100-kbits, it is hard to use the scheme on devices with low resources such as smart cards or RFID tags. This drawback has been recently solved by Gaborit and Girault in [2] where they propose to use the parity check matrix of a random quasi-cyclic code rather than a pure random matrix. This solution permits to preserve the security of the scheme and decreases the description of the random matrix to only a few hundred bits. This new advance opens the door to the use of the Stern protocol in devices with low resources.

Contribution. In this paper we give for the first time a precise description of the implementation of Stern's Protocol (which is very different from the classical number-theory based protocols) and we show how to protect the main steps of the algorithm against side channels attacks. Eventually, we obtain for our implementation an authentication in 6 seconds and a signature in 24 seconds, both without any crypto-processor. This is a promising result when compared to an RSA implementation which would take more than 30 seconds in a similar context without crypto-processor. Stern's Protocol may have a natural application in contexts where the time constraints are not tight such as: authentication for pay-TV or authentication for counterfeiting of expensive goods (*e.g.* ink cartridges of copy machines or expensive clothes). Besides this, the protocol has also the 4 following advantages:

- 1) it can be an alternative to the number-theory based protocols in case a putative quantum computer may exist;
- 2) since it essentially involves linear operations, the protocol seems easier to protect against side channel attacks than the number-theory based protocols;
- 3) the linear operations (scalar products or bit-permutations) are easy to implement in hardware and are very efficient in this context;
- 4) the secret key is smaller than the one of the other protocols (a few hundred bits) for the same security level.

Organisation of the Paper. The paper is organized as follows. In Section 2, we describe Stern's Authentication and Signature schemes and we precise the four main steps of the implementation. In Section 3, we present the side channel attacks in the coding theory context and in Section 4 we propose a secure version of our implementation against side channel attacks. In Section 5, we comment the implementation and eventually we conclude.

2 Stern Authentication Scheme

2.1 Basic Scheme

Stern's Scheme (see [15] for more details) is an interactive zero-knowledge protocol which aims at enabling any user (usually called *the prover P*) to identify himself to another one (usually called *the verifier V*). Let n and k be two integers

such that $n \geq k$. Stern's Scheme assumes the existence of a public $(n - k) \times n$ matrix \tilde{H} defined over the field \mathbb{F}_2 and the choice of an integer $t \leq n$. The matrix \tilde{H} and the weight t are protocol parameters and may be used by several (even numerous) different provers.

Each prover P receives a n -bit secret key s_P (also denoted by s if there is no ambiguity about the prover) of Hamming weight t and computes a *public identifier* i_V such that $i_V = \tilde{H}s_P^T$. This identifier is calculated once in the lifetime of \tilde{H} and can thus be used for several authentications. A user P can prove to V that he is the person associated to the public identifier i_V , by performing the following protocol, (for h a standard hash function):

1. [Commitment Step] P randomly chooses $y \in \mathbb{F}_2^n$ and a permutation σ defined over \mathbb{F}_2^n . Then P sends to V the commitments c_1 , c_2 and c_3 such that :

$$c_1 = h(\sigma|\tilde{H}y^T); \quad c_2 = h(\sigma(y)); \quad c_3 = h(\sigma(y \oplus s)),$$

where $h(a|b)$ denotes the hash of the concatenation of the sequences a and b .

2. [Challenge Step] V sends $b \in \{0, 1, 2\}$ to P .
3. [Answer Step] Three possibilities:
 - if $b = 0$: P reveals y and σ .
 - if $b = 1$: P reveals $(y \oplus s)$ and σ .
 - if $b = 2$: P reveals $\sigma(y)$ and $\sigma(s)$.
4. [Verification Step] Three possibilities:
 - if $b = 0$: V verifies that c_1, c_2 have been honestly calculated.
 - if $b = 1$: V verifies that c_1, c_3 have been honestly calculated.
 - if $b = 2$: V verifies that c_2, c_3 have been honestly calculated, and that the weight of $\sigma(s)$ is t .
5. Iterate the steps 1,2,3,4 until the expected security level is reached.

Fig. 1. Stern's Protocol

Based on the difficulty of the SD problem, it is proven that the protocol is zero-knowledge with a probability of cheating of $(2/3)$ for one round. An appropriate confidence level is reached by repetition of the protocol.

Remark 1. By using the so-called Fiat-Shamir Paradigm [1], it is theoretically possible to convert Stern's Protocol into a signature scheme, but then the signature is very long: about 140-kbit long for 2^{80} security.

Despite the advantages of the protocol (it can be an alternative to number theory based protocol, it is fast and it uses simple linear operations), Stern's Scheme has rarely been used since its publication in 1993. Indeed, the scheme presents the two following drawbacks, which together makes it unpracticable in many applications: 1) many rounds are required (typically 28 if we want the cheater success probability to be less than 2^{-16}), 2) the public key element \tilde{H} is very large (typically 150-kbit long).

The first point is inherent to interactive protocols and in some situations, it does not really constitute a drawback. For instance, if the prover and the

verifier entities can be connected during a long period, then authentication can be achieved gradually. In this case the entire authentication process is performed by executing, time to time during a prescribed period (*e.g.* one hour), an iteration of Algorithm 1 until the expected level of security is reached. Such kind of *gradual authentication* may be of practical interest in pay TV or in systems where a machine (*e.g.* a copy machine or a coffee dispenser) wants to authenticate a physical resource (*e.g.* an ink or a coffee cartridge).

The second drawback has been recently considered by Gaborit and Girault in [2]. We recall the outlines of their approach in the next section.

2.2 Alternative Scheme Based on Quasi-Cyclic Codes

The idea of [2] is to replace the random matrix \tilde{H} by the parity matrix of a particular type of codes whose representation is very compact: the *quasi-cyclic* codes. Let l be an integer value, the parity matrix H of a $[2l, l, \cdot]$ quasi-cyclic code takes the form $H = (I|A)$, where I denotes the $l \times l$ identity matrix and A is a *circulant matrix*, that is a matrix defined for every $(a_1, a_2, a_3, \dots, a_l) \in \mathbb{F}_2^l$ by:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_l \\ a_l & a_1 & a_2 & \dots & a_{l-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_2 & a_3 & a_4 & \dots & a_1 \end{pmatrix} .$$

As it can be easily checked, representing H does not require to store all the coefficients of the matrix (as it is the case in the original Stern’s Scheme) but requires only the l -bit vector $(a_1, a_2, a_3, \dots, a_l)$ (which is the first row of A). Let n equal $2l$, when replacing the random matrix by a random double-circulant one, the parameter sizes of Stern’s Scheme become:

Private data: the secret s of bit-length n .

Public data: the public syndrome i_V of size $\frac{n}{2}$ and the first row of A of size $\frac{n}{2}$, which results in n bits.

It is explained in [2] that for this kind of matrices it is enough to take $l = 347$ and $t = 74$ (and hence $n = 694$). As the new version of Stern’s Scheme involves parameters with small sizes and continues to use only elementary logical operations, it becomes highly attractive for light-weight implementations. This is especially true for environments where memory (RAM, PROM, etc.) is a rare resource. The version of Stern Scheme discussed in the rest of the paper involves a $\frac{n}{2} \times n$ *double circulant matrices*.

2.3 Main Operators

A quick analysis of Stern’s Protocol shows that the different steps are merely composed of the four following main operators:

Matrix-vector product: the multiplication of a vector by a random double circulant matrix;

Hash function: the action of a hash function;

Permutation: the generation and the action of a random permutation on words;

PRNG: a pseudo-random generator used to generate random permutations and random vectors.

By using quasi-cyclic codes, it becomes possible to implement the prover application of Stern's Scheme in an embedded device (as *e.g.* a smartcard). However, being implemented in a low resource device, the prover application becomes vulnerable to side-channel attacks and appropriate countermeasures must therefore be added. In the two following sections, we discuss about the problematic of side-channel attacks in our context and then, we precise for each of the above operators a way to implement it and how to protect it against side-channel attacks.

3 Side-Channel Attacks

Side-channel attacks aim at recovering information about *sensitive variables* appearing in the description of the algorithm under attack. We shall say that a variable is sensitive if it is a function of both public data and of a secret (resp. private) parameter of the algorithm.

In the protocol we described in Fig. 1, we only want to implement Steps 1 and 3 (the ones that rely on the prover) in an embedded device. The other Steps 2 and 4 (that rely on the verifier) may indeed be performed on a PC. For Steps 1 and 3, we have the following list of sensitive variables that can be potentially targeted by SCA:

Threat A. the random vector y in the computations of c_1 (when performing $\tilde{H}y^T$) and of c_2 (when performing $\sigma(y)$): if an attacker is able to retrieve y during one of these steps, then with a probability $1/3$ he is able to recover s when A answers $y \oplus s$ to the ($b = 1$)-challenge.

Threat B. the private vector s during the computation $\sigma(s)$: in this case the attacker recovers A 's private parameter.

Threat C. the private vector s during the computation $\sigma(y \oplus s)$: in this case the attacker recovers A 's private parameter.

Threat D. the bit-permutation σ during the computation of $\sigma(s)$ or $\sigma(y \oplus s)$: if an attacker is able to retrieve σ during one of these steps, then with a probability $1/3$ he is able to recover s when A answers $\sigma(s)$ to the ($b = 2$)-challenge.

Threat E. the bit-permutation σ during the computation of the hash value $h(\sigma|\tilde{H}y^T)$: if an attacker is able to retrieve σ , then with a probability $1/3$ he is able to recover s when A answers $\sigma(s)$ to the ($b = 2$)-challenge.

Remark 2. When looking for sensitive variables in Fig. 1, we have assumed that the analysis of the device behavior during the storage or the loading of a data does not bring useful information about it. In other terms, we made the classical assumption that information about a data only leaks from calculus involving it

(and eventually other public information) and that data manipulations themselves do not leak enough information on current devices.

To retrieve information about the sensitive data listed above, we assume in the rest of the paper that the SCA adversary can only perform an attack belonging to one of the following three categories:

1. The so-called *timing* attacks consist in analyzing the time taken to execute cryptographic algorithms.
2. The so-called *simple analysis* attacks (SPA in short) are on-line attacks that consist in directly interpreting power consumption measurements and in identifying the execution sequence.
3. The so-called *correlation* attacks (DPA in short) work as *greedy* algorithms: the side-channel information is analyzed by statistical means until the secrets are extracted.

It may be noticed that other categories of SCA attacks exist as for instance the templates or the Higher Order SCA ones. We chose to not consider these attacks for our implementation since they rely on a much stronger (and hence less realistic) adversary than the ones involved in the attacks listed above. For more details about SCA, the reader is referred to [7].

Let us now present the outlines of the defense strategy we shall apply to protect the implementation of the algorithm described in Fig. 1.

Defense Strategy. To deal with timings attacks issue, both hardware and software countermeasures are usually involved simultaneously. At the software level, a classical defense strategy consists in implementing all the operations involving sensitive data in a way that does not depend on the data value (for instance methods based on conditional branches are precluded). We chose to follow this strategy for all the operations that are susceptible to manipulate sensitive data.

The most common way of thwarting SPA and DPA involves random values (called *masks*) to de-correlate the leakage signal from the sensitive data which are manipulated. This protection method is usually called *first order masking*. It has been argued in several recent papers (e.g. [3,9,14]) that this method is sound (when combined with usual hardware protections) to protect an algorithm against SPA and all kinds of first order DPA.

In a first order masking of an algorithm, every sensitive variable y appearing in the algorithm is never directly manipulated by the device and is represented by 2 values \tilde{y} (the *masked data*) and M (the *mask*). To ensure the DPA-resistance, the mask M takes random values and to ensure completeness, \tilde{y} satisfies

$$\tilde{y} = y \oplus M \quad . \quad (1)$$

Since y is sensitive, every function S of y is also sensitive as long as S is known by the attacker. Let z denote this new sensitive value $S(y)$. To mask the processing of z without revealing information on y , two new values \tilde{z} and N must

be computed from (\tilde{y}, M) (which represents y in the implementation) in such a way that

$$\tilde{z} \oplus N = z = S(y) . \quad (2)$$

The critical point of such a method is to deduce the new pair of (masked value)/mask (\tilde{z}, N) from the previous pair (y, M) without compromising the security of the scheme with respect to first order DPA. This problem is usually referred as the *mask correction Problem*.

When S is linear, it can be resolved very efficiently since we have:

$$z = S(y) = S(y \oplus M \oplus M) = S(\tilde{y}) \oplus S(M) , \quad (3)$$

Hence, we simply have to define \tilde{z} and N such that $\tilde{z} = S(\tilde{y})$ and $N = S(M)$.

Dealing with the mask correction Problem when S is non-linear is much more difficult. Numerous papers have been published which aim to tackle this issue (an overview of the existing methods is proposed in [14]). As argued in [14], when the input and output dimensions n and m of the function S are small, then the so-called *Re-computation* method (REC in short) is the most appropriate one since it only requires one memory transfer and the pre-processing of a RAM table of 2^n elements (one time per algorithm execution):

Re-computation method. Let M and N be two random variables and let us assumed that the RAM look-up table S^* associated to the function $y \mapsto S(y \oplus M) \oplus N$ has be pre-processed. Then, to compute $S(y) \oplus N$ from $\tilde{y} = y \oplus M$, the REC method performs a single operation: the table look-up $S^*[\tilde{y}]$.

As we will see in the next sections, applying first order masking to Stern's Protocol induces only a very small timing overhead and an acceptable memory overhead, since almost all the performed operations are linear (and thus Relation (3) applies most of the time). Moreover, for the few non-linear operations that must be protected (in particular when the bit-permutation σ is computed), we can apply efficiently the REC method since the dimensions of the involved sub-functions are small.

4 Algorithm Specification

In this following, we focus on the four operators defined in Section 2. For each of them, we exhibit an efficient implementation and we discuss about how to protect it effectively against SPA and DPA.

4.1 Matrix-Vector Product

Algorithm Description For a Quasi-cyclic Matrix. When double-circulant matrices are involved, very efficient algorithms exist to compute the matrix-vector product. In the following, we detail the computation of the product $\tilde{H} \times v$ between the $\frac{n}{2} \times n$ double-circulant matrix $\tilde{H} = (I|A)$ and the n -bit vector v . In our description, we shall denote by *matrix* the $\frac{n}{2}$ -bit row vector of A and by *result* an $\frac{n}{2}$ -bit temporary vector. Also, we shall denote by $|reg|$ the number of

bits contained in a register of the processor and by $nblocs$ the number of blocs in *matrix*: $nblocs = \lceil \frac{n}{2 \times |reg|} \rceil$. Additionally, we will denote by v_L (resp. by v_R) the least significant half part of v (resp. the most significant half part of v): namely, we have $v_L = (v_1, \dots, v_{n/2})$ and $v_R = (v_{n/2+1}, \dots, v_n)$.

Algorithm 1. Quasi cyclic matrix vector product

INPUT: $matrix = \tilde{H}, v, |reg|$

OUTPUT: $result = \tilde{H}v^T$

1. **for** i **from** 1 to $\frac{n}{2}$ **do** $result[i] = v_L[i]$; // initialisation with the first half of the vector
 2. **for** i **from** 1 to $|reg|$ **do**
 3. **if** $i > 1$; v_R is rotated of one bit to the left;
 4. **for** j **from** 1 to $|nblocs|$ **do**
 5. **if** the i -th bit of $matrix[j] == 1$; // add v_R to the result beginning with the j th bloc
 6. $br = 1$
 7. **for** jj **from** j to $|nblocs|$ **do**
 8. $result[br] = result[br] \oplus v_R[jj]$; $br = br + 1$;
 9. **for** jj **from** 1 to $j - 1$
 10. $result[br] = result[br] \oplus v_R[jj]$; $br = br + 1$;
 11. **return** $result$
-

SCA-Security Discussion. As argued in Section 3, information about the sensitive data y may leak during the matrix-vector product $\tilde{H}y^T$ (Threat A) and a first order masking must thus be applied. As this product is linear for the bitwise addition and due to (3), masking the calculus is straightforward and implies an acceptable timing/memory overhead.

Before computing $result = \tilde{H}y^t$, the vector y is masked with a n -bit mask M (randomly generated). Then Algorithm 1 is input with *matrix* (i.e. the first row of A) and $\tilde{y} = y \oplus M$. The corresponding output is $\tilde{H}\tilde{y}^t = result \oplus N$, where we denoted by N the value $\tilde{H}M^t$. As all the coordinate-bits of y are masked with a uniformly distributed random value, the SPA or the first order DPA analysis of the matrix-vector product does not bring information about y . To make the future unmasking of \tilde{y} possible, a second matrix-vector product $N = \tilde{H}M^T$ is performed and stored in memory together with \tilde{y} .

Complexity Discussion : Secure Version. For a quasi-cyclic matrix of size $\frac{n}{2} \times n$ whose first row is of weight $p+1$, the following steps have to be undertaken:

- masking the matrix $\tilde{y} = y \oplus M$.
- computing $\tilde{H}\tilde{y}^t = result \oplus N$, where $N = \tilde{H}M^t$.
- a second matrix-vector product $N = \tilde{H}M^T$ is performed and stored in memory together with \tilde{y} .
- extract and test the $n/2$ bits of the matrix first row for the product $\tilde{H}\tilde{y}^t$
- extract and test the $n/2$ bits of the matrix first row for the product $\tilde{H}M^t$
- $\lceil \frac{n}{2 \times |register|} \rceil$ binary-shifts of the vector \tilde{y}

- $\lceil \frac{n}{2 \times \text{register}} \rceil$ binary-shifts of the vector M
- $2p \times \lceil \frac{n}{2 \times \text{register}} \rceil$ registers to be added to the two results

The secure version requires two products matrix vector one for the mask and one for the product to determine. The cost is therefore around the double of the one of the non-secure version.

4.2 Hash Function

To counteract Threat E, the Stern Protocol Implementation must involve a hash function implementation that is secure against first order DPA. Until now, the securing of hash function implementations against SCA has been rarely focused, essentially because these functions usually operate on non-sensitive (often public) data. However, Lemke *et al.* [4] or McEvoy *et al.* [8] have shown that, in some applications like HMAC authentication, mounting DPA attacks against hash functions makes sense when secret (or private) data have to be hashed together with public data. In [8], the authors exhibit a way to protect an hardware implementation of the hash function SHA-256 against first order DPA. In the rest of this section, we will assume that the device on which is implemented the Stern Protocol possesses such a secure hardware implementation of SHA-256. It may be noticed that the masking method used by McEvoy *et al.* for hardware implementations may also be followed to design a masked software implementation of SHA-256. However, in this case, the timing and memory overheads become too large. Actually, if the device does not have a secure SHA-256 implementation, it may be pertinent to use a hash function based on block cipher constructions (the State of the Art of hash functions published by Preneel in [13] give several examples of such functions). Indeed, in such a case the hash function can inherit the DPA-security from the involved block cipher algorithm and the nowadays embedded devices possess almost always a DES Hardware and sometimes an AES Hardware that include anti-DPA mechanisms. In the case where neither secure hash function nor secure block cipher algorithms are implemented in the device, then it is always possible to use one of the numerous DES or AES DPA-secure software implementations proposed in the Literature (see for instance [9],[14]) and to involve them in a hash function based on block ciphers (like for instance MDC-4).

Remark 3. For a hash function to provide a satisfying security, the bit-length of the hash values it produces must be at least 160. When using hash functions based on block ciphers, it may be difficult to get hash values of such a bit-length. In this case, a solution may be to concatenate several output blocks until the bit-length 160 is achieved or exceeded and, if necessary, to truncate in order to get a length of exactly 160 bits (for instance two AES ciphering will result in 256 bits which can be truncated to 160 bits).

Algorithm Description : Secure Hardware Implementation of SHA-256. In [8], McEvoy *et al.* describe a first order masked hardware implementation of the HMAC algorithm based on SHA-256. Using an implementation on

a commercial FPGA board, they present a masked hardware implementation of the algorithm, which is designed to counteract first-order DPA attacks.

SCA-Security Discussion. In [8], the resistance of the SHA-256 implementation is formally analyzed and demonstrated.

Complexity Discussion. It is shown in [8] that the processor and the interface circuitry corresponding to the masked SHA-256 utilize 1734 slices (37% of the FPGA resources) and that the critical path in the design (i.e. the longest combinational path) is 18.6. As argued in [8], the area has almost doubled compared with the unprotected implementation but the speed has not been overly affected.

4.3 Permutation Method

Defining a vectorial permutation σ over \mathbb{F}_2^n (like the one used in Figure 1) amounts to define an *index permutation* ψ over $\{0, \dots, n-1\}$ such that for every $y = (y[0], \dots, y[n-1]) \in \mathbb{F}_2^n$ we have $\sigma(y) = (y[\psi(0)], \dots, y[\psi(n-1)])$. In this paper, we chose to design the permutation ψ by following the approach suggested by Luby and Rackoff in [5,6] and improved in [10,12]. The core idea of this approach is to involve a few pseudo random functions in a Feistel Scheme. As argued by the authors in [10,12], such a method makes it possible to design random permutations very efficiently since only a few Feistel rounds are needed and since the input/output dimensions of the involved functions are more or less logarithmic in the size of the words on which the permutation operates.

Let us first recall some basic facts about the so-called Luby-Rackoff schemes.

Luby-Rackoff's Scheme. For every function f defined from \mathbb{F}_2^l into \mathbb{F}_2^m , the Feistel round involving f , denoted by $\psi(f)$, is defined for every pair $(L, R) \in \mathbb{F}_2^m \times \mathbb{F}_2^l$ by $\psi(f)[L, R] = [R, L \oplus f(R)]$. The composition of k Feistel rounds, that is the function $\psi(f_k) \circ \dots \circ \psi(f_1)$, is denoted by $\psi(f_1, \dots, f_k)$ or by ψ in short if there is no ambiguity about the involved functions.

If f and g are two randomly generated independent functions defined from \mathbb{F}_2^m into itself, then it has been argued in [10,12] that the function $\psi(g, f, g, f)$ is indistinguishable from an uniform distribution by an observer, even if the latter has access to the inverse permutation. As a consequence, to design an index-permutation ψ over $\{0, \dots, 2^{2m}-1\}$ (and thus a vectorial bit-permutation σ over \mathbb{F}_2^n with $\log_2(n) = 2m$), we simply need to generate the two independent random functions f and g .

Once the Luby-Rackoff scheme has been designed for two functions f and g , there are merely two strategies to compute the bit-permutations $\sigma(y)$, $\sigma(y \oplus s)$ and eventually $\sigma(s)$ in Stern's Protocol. The first one consists in pre-computing $\psi(i)$ for every $i \leq n$ and then to store the sequence $(\psi(i))_{i \leq n}$ as a representation of σ . In this case, each time σ must be applied to a vector, then its table representation is accessed n times (one time for each bit-index). This strategy requires the RAM allocation of $n \times \lceil \log_2(n) \rceil$ bits, which is quite expensive in a low resource context. The second strategy consists in computing $(\psi(i))_{i \leq n}$ each

time one needs to determine the bit index corresponding to i in σ . This strategy, which has been chosen for our implementation, is more time consuming than the previous one but it does not require any RAM allocation.

By construction, Luby-Rackoff Scheme only permits to construct index permutations ψ such that n is a power of 2. Since the size parameter n we consider for Stern's Scheme is 347 (see Section 2), we couldn't use Luby-Rackoff Scheme straightforwardly, but a slightly modified version of it.

Algorithm Description. In Section 2 we argued that the parameters size (of s and y) should be at least $n = 694 = 2 \times 347$. Let m denote the value $\lceil \log_2(n) \rceil / 2$. To implement a permutation on vectors of any bit-length n such that $\lceil \log_2(n) \rceil$ is even, we suggest hereafter to randomly generate two functions f and g defined from \mathbb{F}_2^m into itself and to use the Luby-Rackoff Scheme in the following way:

Algorithm 2. Bit-permutation for any n such that $\lceil \log_2(n) \rceil$ is even
 INPUT: the vector v to permute, the bit-length n of v , the value $n' = 2^{\lceil \log_2(n) \rceil}$, a Luby-Rackoff Scheme $\psi(g, f, g, f)$ with f and g defined from $\mathbb{F}_2^{\lceil \log_2(n) \rceil / 2}$ into itself.
 OUTPUT: the vector $result = \sigma(v)$

```

1. for  $i$  from 0 to  $n' - 1$  do  $T[i] \leftarrow 0$ 
2. for  $i$  from 0 to  $n' - 1$  do
3.    $new\_index \leftarrow \psi(i)$ ;  $result[new\_index] \leftarrow v[i]$ ;  $T[new\_index] \leftarrow 1$ ;
4.  $j \leftarrow n$ ;
5. for  $i$  from 0 to  $n - 1$  do
6.   if ( $T[i] = 0$ )
7.     while ( $T[j] = 0$ ) do  $j \leftarrow j + 1$ ;
8.      $result[i] \leftarrow result[j]$ ;  $j \leftarrow j + 1$ 
9. return  $result$ 

```

Remark 4. Table T needs to be computed only one time per each permutation σ . Once computed, it can be used for all the permutation involving σ .

In Algorithm 2, each iteration of the second loop computes the bit-index new_index in $result$ where to store the bit-value $v[i]$. During this processing, Table T keeps trace of the $result$ bit-coordinates that are updated during this process. When the loop is ended, a third loop is iterated to fill the bit-coordinate of index $i < n$ that has not been initialized by the second loop (which are the ones such that $T[i] = 0$), with the bit-coordinates of index $n \leq j < n'$ that has been initialized (which are such that $T[j] = 1$).

SCA-Security Discussion. In order to thwart Threats C and D (see Section 3), we chose to mask the computations of σ and ψ . Since σ is at most used 3 times before being re-generated, it may be targeted by SPA attack but does not suffer from DPA. The linearity of σ makes it easy to mask its processing: we mask the input y with a random mask M (which results in a masked input $\tilde{y} = y \oplus M$) and we unmask the output $\sigma(\tilde{y})$ by simply x-oring it with $\sigma(M)$.

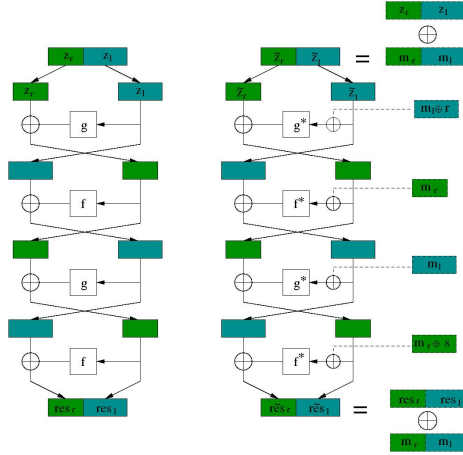


Fig. 2. Luby Rackoff permutation - unsecure and secure versions

In Algorithm 2, the same function ψ is applied $2^{n'}$ times on known input before being re-generated. It can thus be targeted by DPA attacks. To counteract them, we chose to mask the intermediate variables that appear during the processing of ψ and to apply the REC method to deal with the mask correction problem when the functions f and g are used. Each time new functions f and g are generated (thus defining a new function ψ), we generate two random masks $r, s \in \mathbb{F}_2^m$ and we define two new functions f^* and g^* such that $f^*(x) = f(x \oplus r) \oplus s$ and $g^*(x) = g(x \oplus s) \oplus r$. We describe the normal and secure processing of ψ in Figure 2.

Complexity Discussion. The function ψ must be re-generated at each execution of Algorithm 2. This requires the generation of $2 \times m \times 2^m$ random bits to define the functions f and g . The un-secure processing of ψ involves 4 look-up tables and 4 bitwise m -bit additions. Its secure processing requires the pre-computation of the new RAM lookup-tables f^* and g^* (with complexity $O(2^m)$) and 8 additional bitwise m -bit additions (4 for the mask correction and 2×2 for the masking/unmasking of the input/output) compared to the un-secure calculus.

Algorithm 2 involves two n' -bit local variables T and $result$. It processes n' times the function ψ and executes two loops involving around respectively $2 \times n'$ and $4 \times n'$ elementary operations. This results, for the σ processing, in $14n' = 8 \times n' + 2 \times n' + 4 \times n'$ elementary operations and in the generation of $2 \times m \times 2^m$ random bits for the processing of ψ without any SCA countermeasure. In the SCA-secure mode, the processing of σ requires $22n' = (8+8) \times n' + 2 \times n' + 4 \times n'$ elementary operations and the generation of $2 \times m \times 2^m$ random bits for f and g , one n -bit vectors M to mask the input y of σ and one $2m$ -bit vector (m_r, m_l) to mask all the input of ψ . To make the final unmasking of $\sigma(y \oplus M)$ possible, the vector $\sigma(M)$ must also be computed, which adds $22n'$ elementary

operations (note that the mask (m_r, m_l) does not need to be re-generated to protect the processing of ψ for $\sigma(M)$). Finally, we get for the secure processing in the secure mode, around $44n'$ (i.e. $44 \times 2^{\lceil \log_2(n) \rceil}$) elementary operations and the generation of $n+2 \times m \times 2^m + 2m$ random bits (i.e. $n + \lceil \log_2(n) \rceil \times 2^{\lceil \log_2(n) \rceil / 2} + \lceil \log_2(n) \rceil$).

Example 1. For the choice of parameter size done in Section 2 (i.e. $n = 694$), we have $m = \lceil \log_2(n) \rceil / 2 = \lceil \log_2(694) \rceil / 2 = 5$ and $n' = 2^{\lceil \log_2(n) \rceil} = 2^{10} = 1024$. In such a case, the processing of σ without any security requires around $14 \times 10^3 \approx 14 * 1024$ elementary operations and the generation of $320 = 10 \times 32$ random bits. In the SCA-secure mode, it requires around $45 \times 10^3 \approx 44 * 1024$ elementary operations and the generation of $1024 = 694 + 10 \times 32 + 10$ random bits.

Example 2. For $n = 512$ (which is the choice of parameter size done in Section 5), the processing of σ without any security requires around $7 \times 10^3 \approx 14 * 512$ elementary operations and the generation of $224 = 4 \times 2^4 + 5 \times 2^5$ random bits (note that in this case, since $\log_2(512) = 9$ is odd, the functions f and g cannot have the same dimensions and we chose f being from \mathbb{F}_2^4 into \mathbb{F}_2^5 and g being from \mathbb{F}_2^5 into \mathbb{F}_2^4). In the SCA-secure mode, it requires around $22 \times 10^3 \approx 44 * 512$ elementary operations and the generation of around 450 random bits.

4.4 Pseudorandom Generator

We need a pseudorandom generator to construct the seed of the code and the permutation. Nowadays, most of the pseudo-random generators used in commercial applications are either based on stream cipher or on block cipher algorithms. Hardware implementations of stream cipher are often faster than the ones of block ciphers. However, there are only available in some specific devices (and are for instance not available in most of the smart cards), whereas block ciphers algorithms such as DES or AES are almost systematically implemented in hardware. We consider that the Pseudo Random Number Generator (PRNG) that we will use to generate the seed is not biased and secure against SPA and DPA attacks. In [11], the author present a block cipher-based PRNG secure against side-channel key recovery.

5 Implementation

5.1 Experimental Results

We have realized the implementation of Stern Authentication with double circulant matrices for $l = 256$ (i.e. $n = 512$) on a 8051-architecture **without crypto-processor** nor hardware SHA-256, and with a CPU running at 12 MHz.

Remark 5. Timing performances given in Table 1 do not take the communication cost into account. This choice has been made because the transmission rate highly depends on the application type. For instance, the today VISA norm

Table 1. Performances of the implementation

Time for 1 round	Time (ms)
PRNG (vector y , function f and g)	16.7
Matrix-vector product	22.0
Permutations (and an xor)	22.6
Hash function (SHA-256)	107.6
Total for one round	168.9
Authentication (35 rounds)	5 911.5

imposes 9600 bauds (which is quite low), whereas the nowadays technologies make it possible to have 110000 bauds for transmission rate.

We obtain an authentication in ≈ 6 seconds and a signature in ≈ 24 seconds for a security of 2^{85} . The communication cost is around 40-kbits in the authentication scheme and around 140-kbits for the signature. It must be noticed that the timing performances would be highly improved by using a hardware SHA-256 instead of a software implementation.

The implementation detailed above doesn't include SCA countermeasures. According to the study conducted in Section 4, the timing/memory overhead expected after securization is around ($\times 3$). This value is really small compared for instance to a secure software version of the AES where the overhead is around $\times 10$.

6 Conclusion and Future Work

We have described in this paper the first implementation of Stern protocol on smart card (in fact it is also more generally the first code-based system implemented on smart-card with usual resources). For a satisfying security level, the size of the public key is only 694 bits using a quasi cyclic representation of the matrix considered. The double-circulant matrices are a good trade-off between random and strongly structured matrices. In this case the operations are indeed really simple to perform and can be implemented easily in hardware. Moreover, the fact that the protocol essentially performs linear operations makes the algorithm easy to protect against side channel attacks. We thus think that the protocol is a new option to carry out fast strong authentication on smart cards. Additionally, we think that the use of a dedicated linear-algebra co-processor should significantly improve the timing performances of our implementation.

Future work besides this one includes considering Fault injection attacks (this was beyond the scope of this paper) and implementation of other variations of Stern protocol which can have other small advantages (see [2]) for protocol variations.

References

1. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odyszko, A. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987)
2. Gaborit, P., Girault, M.: Lightweight code-based identification and signature. In: IEEE Transactions on Information Theory (ISIT), pp. 191–195 (2007)
3. Goubin, L., Patarin, J.: DES and Differential Power Analysis – The Duplication Method. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
4. Lemke, K., Schramm, K., Paar, C.: DPA on n-bit sized boolean and arithmetic operations and its applications to IDEA. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 205–219. Springer, Heidelberg (2004)
5. Luby, M., Rackoff, C.: Pseudo-random permutation generators and cryptographic composition. In: Symposium on Theory of Computing, vol. 18, pp. 353–363 (1986)
6. Luby, M., Rackoff, C.: How to construct pseudorandom permutation and pseudorandom functions. SIAM J. Comput. 17, 373–386 (1988)
7. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks – Revealing the Secrets of Smartcards. Springer, Heidelberg (2007)
8. McEvoy, R., Tunstall, M., Murphy, C., Marnane, W.P.: Differential power analysis of HMAC based on SHA-2, and countermeasures. In: Kim, S., Yung, M., Lee, H.-W. (eds.) WISA 2007. LNCS, vol. 4867, pp. 317–332. Springer, Heidelberg (2008)
9. Oswald, E., Schramm, K.: An Efficient Masking Scheme for AES Software Implementations. In: Song, J., Kwon, T., Yung, M. (eds.) WISA 2005. LNCS, vol. 3786, pp. 292–305. Springer, Heidelberg (2006)
10. Patarin, J.: How to construct pseudorandom and super pseudorandom permutation from one single pseudorandom function. In: Rueppel, R. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 256–266. Springer, Heidelberg (1993)
11. Petit, C., Standaert, F.-X., Pereira, O., Malkin, T.G., Yung, M.: A Block Cipher based PRNG Secure Against Side-Channel Key Recovery, <http://eprint.iacr.org/2007/356.pdf>
12. Pieprzyk, J.: How to construct pseudorandom permutations from single pseudorandom functions advances. In: Damgård, I. (ed.) EUROCRYPT 1990. LNCS, vol. 473, pp. 140–150. Springer, Heidelberg (1991)
13. Preneel, B.: Hash functions - present state of art. ECRYPT Report (2005)
14. Proff, E., Rivain, M.: A Generic Method for Secure SBox Implementation. In: Kim, S., Yung, M., Lee, H.-W. (eds.) WISA 2007. LNCS, vol. 4867, pp. 227–244. Springer, Heidelberg (2008)
15. Stern, J.: A new identification scheme based on syndrome decoding. In: Stinson, D. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 13–21. Springer, Heidelberg (1994)

A Practical DPA Countermeasure with BDD Architecture

Toru Akishita, Masanobu Katagi, Yoshikazu Miyato,
Asami Mizuno, and Kyoji Shibutani

System Technologies Laboratories, Sony Corporation,
1-7-1 Konan, Minato-ku, Tokyo 108-0075, Japan
{Toru.Akishita,Masanobu.Katagi,Yoshikazu.Miyato}@jp.sony.com,
{Asami.Mizuno,Kyoji.Shibutani}@jp.sony.com

Abstract. We propose a logic-level DPA countermeasure called Dual-rail Pre-charge circuit with Binary Decision Diagram architecture (DP-BDD). The proposed countermeasure has a dual-rail pre-charge logic style and can be implemented using CMOS standard cell libraries, which is the similar property to Wave Dynamic Differential Logic (WDDL). By using novel approaches, we can successfully reduce the early propagation effect, which is one of the main factors of DPA leakage of WDDL. DP-BDD is suited to implementation of S-boxes. In our implementations of the AES S-box, DP-BDD can reduce the maximum difference of transition timing at outputs of S-box to about 1/6.5 compared to that of WDDL without delay adjustment. Moreover, by applying simple delay adjustment to the inputs of the S-box, we can reduce it to about 1/85 of that without the adjustment. We consider DP-BDD is a practical and effective DPA countermeasure for implementation of S-boxes.

Keywords: DPA, countermeasure, dual-rail pre-charge logic, Binary Decision Diagram.

1 Introduction

Differential Power Analysis (DPA) is a serious threat to cryptographic devices such as smart cards [8]. Recently, various countermeasures have been proposed to protect implementations of cryptographic algorithms against DPA at the logic level. Since the logic-level countermeasures can be adapted to basic logical gates such as an AND gate, we can apply them to implementations of any cryptographic algorithms. These logic-level countermeasures are classified into the following three groups: masking logics, dual-rail pre-charge logics, and hybrid-type logics.

Masking logics try to randomize the activity at every node in a circuit using random values in order to remove correlation between key-related intermediate values and power consumption of the circuit. Masked-AND, a type of masking logics, was proposed by Trichina [20]. It has been pointed out, however, that Masked-AND is not completely secure due to the effect of glitches [9,14]. Recently, Random Switching Logic (RSL) was proposed by Suzuki et al. [16]. RSL

is theoretically secure under the leakage models described in [14], but possesses two disadvantages: one is that it cannot be implemented using CMOS standard cell libraries and the other is that it requires careful timing adjustment of enable signals.

A dual-rail pre-charge logic was first proposed by Tiri et al. as Sense Amplifier Based Logic (SABL) [17], where a signal is represented by two complementary wires and one of these two wires is charged and discharged in every cycle. Considering that SABL needs a special CMOS library, Tiri et al. also proposed Wave Dynamic Differential Logic (WDDL) [18] that can be implemented using CMOS standard cell libraries. WDDL is a practical countermeasure, but it cannot suppress two factors of DPA leakage. The first one is due to unbalanced load capacitance of complementary logic gates. In order to balance it, WDDL requires a custom layout for a secure design [19,7]. The other is due to the early propagation effect. This leakage is caused when input signals of a WDDL gate have difference of delay time [14]. The input signals generally pass the different number of logic gates, and then the difference of delay time inevitably occurs. Careful delay adjustment can reduce the difference, but applying it all WDDL gates in cryptographic circuits seems to be unrealistic.

Hybrid-type logics are combined with masking logics and dual-rail pre-charge logics. At CHES 2005, Popp and Mangard proposed MDPL that combines dual-rail pre-charge circuits with random masking to improve the first disadvantage of WDDL [11]. They claimed that it can achieve secure design using a CMOS standard cell library without special layout constraint, but in the next year it was pointed out that MDPL can be still insecure when there is relatively large difference in delay time between the input signals of MDPL gates [4,15]. In addition, the combination of masking and dual-rail was shown to be unable to provide a routing-insensitive logic style [6,13]. At present, hybrid-type logics seem to have no advantage over dual-rail pre-charge logics.

In this paper, we propose a novel DPA countermeasure called *Dual-rail Pre-charge circuit with Binary Decision Diagram architecture* (DP-BDD). It is based on a Binary Decision Diagram (BDD) that is a direct acyclic graph used to represent a Boolean function. DP-BDD is composed of AND-OR gates which are included in CMOS standard cell libraries. Due to the based BDD architecture, the input signals of an AND-OR gate always pass the same number of AND-OR gates, and then the early propagation effect, which is one of the main factors of DPA leakage of WDDL, is significantly reduced.

This DPA countermeasure is suited to implementation of S-boxes. In our implementations of the AES [10] S-box, DP-BDD can reduce the maximum difference of transition timing at the outputs of the S-box to about 1/6.5 compared to that of WDDL without delay adjustment. Moreover, by applying simple delay adjustment to the inputs of the S-box, we can reduce it to about 1/85 of that without the adjustment. DP-BDD requires a custom layout to prevent the leakage caused by unbalanced load capacitance of complementary logic gates the same as WDDL, but we consider that DP-BDD is a practical and effective DPA countermeasure for implementation of S-boxes.

The rest of the paper is organized as follows: Section 2 presents WDDL and its security problem. Section 3 gives brief introduction of BDD that is the basic architecture of our method. In Section 4 we present the proposed DPA countermeasure called DP-BDD. In Section 5, we apply WDDL and DP-BDD to implementations of AES S-box and compare their effectiveness. We introduce simple delay adjustment of DP-BDD to reduce the difference of transition timing further in Section 6. Finally we draw our conclusion and discuss further work in Section 7.

2 Wave Dynamic Differential Logic (WDDL)

Tiri et al. proposed Wave Dynamic Differential Logic (WDDL) as a logic-level countermeasure of DPA [18]. WDDL has the following features:

- WDDL gates have complementary inputs and outputs.
- WDDL has the pre-charge phase to transmit $(0, 0)$ and the evaluation phase to transmit $(0, 1)$ or $(1, 0)$, and performs these phases mutually.
- WDDL can construct combinational logics by using only AND gates, OR gates, and NOT operations (signal swapping).

A value a is represented (a, \bar{a}) in WDDL, where \bar{a} is the negation of a . An activity factor within WDDL circuits is constant independent of the input signals due to the above features. Since power consumption at CMOS gates generally depends on the transition probability of the gates, WDDL is considered to be effective as a DPA countermeasure.

However, the power consumption at CMOS gates also depends on load capacitance of the gates. If there is difference of load capacitance between complementary logic gates of WDDL, the difference of power consumption occurs. The number of gates connected to complementary logic gates of WDDL is basically equal, and then the difference of load capacitance is caused by the difference of place-and-route. The leakage due to the place-and-route is called as *incidental leakage* [15]. It can be improved by the place-and-route in the manual or semi-automatic operation using special constraints such as “Fat Wire” [19] and “Backend Duplication” [7].

Another leakage is due to the early propagation effect [14, 15]. This leakage is caused when there is the difference of delay time between the input signals of a WDDL gate. In Fig. 1, we illustrate a WDDL AND gate and its signal transitions according to the inputs (a, b) . Here, we assume that the transition of a or \bar{a} reaches the gate earlier than the transition of b or \bar{b} both on the evaluation phase and on the pre-charge phase. The transition timing of the complementary output q or \bar{q} on the evaluation phase depends on the input a . On the other hand, the transition timing of q or \bar{q} on the pre-charge phase depends on the input b . Therefore, the difference of delay time between the inputs a and b may leak the values a and b . Since basic cryptographic components including S-boxes

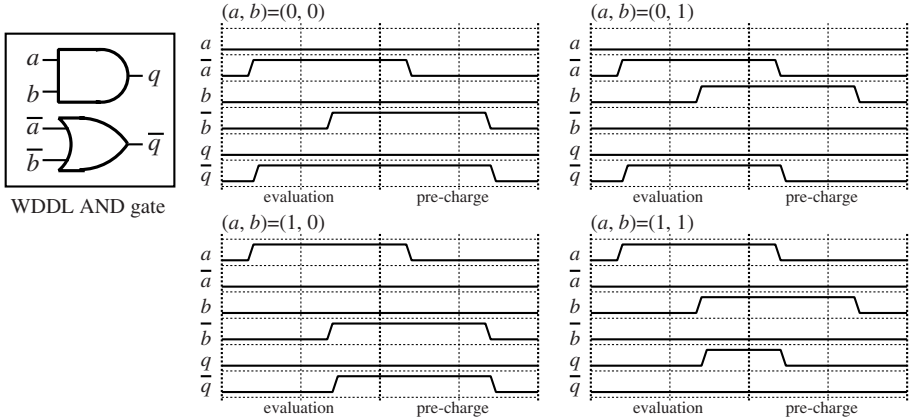


Fig. 1. The early propagation effect of a WDDL AND gate

of blockciphers require complicated logic circuits, the input signals of a WDDL gate generally pass different number of logic gates. Therefore, the difference of delay time between these signals inevitably occurs. This type of leakage is called as *inevitable leakage* [15]. The leakage can be improved by adjusting delay time between the input signals, but very high effort and many constraints in the circuit design are required to adjust delay time of all WDDL gates in complicated logic circuits including S-boxes.

3 Binary Decision Diagram

A Binary Decision Diagram (BDD) is a direct acyclic graph that is used to represent a Boolean function [1], and one of most commonly used synthesis tools for logic optimization of digital systems [22]. We briefly explain a BDD according to Fig. 2. The left figure is a truth table representing the function $f(A, B, C)$ and the right figure shows a block diagram of a binary decision tree corresponding to the truth table. In the right figure, an isosceles trapezoid represents a 2-to-1 multiplexer, and we call a signal A, B, C as a *non-terminal node*, a signal $0, 1, 0, \dots$ at the lowest part as a *terminal node*, and a signal connecting two multiplexers as an *internal node*. The outputs f in the truth table are located in regular order from the left to the right of terminal nodes.

Generally the term BDD refers to Reduced Ordered Binary Decision Diagram (ROBDD) [2]. A binary decision tree is uniquely transformed into ROBDD by merging any isomorphic subgraphs and eliminating any redundant nodes. In this paper, however, we call as BDD the block diagram in which we only merge any isomorphic subgraphs on a binary decision tree. In this BDD architecture, since the same number of multiplexers must be passed from any terminal node to the output, the difference of propagation delay dependent of inputs is relatively small.

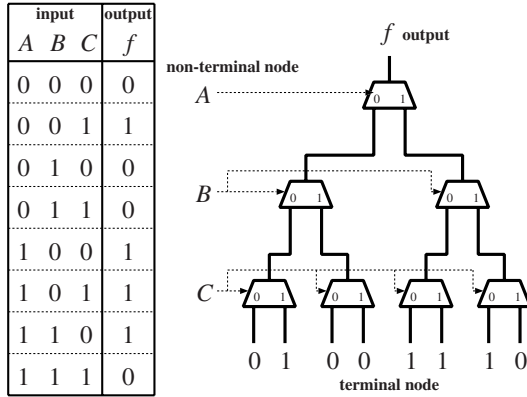


Fig. 2. A truth table and a binary decision tree

4 Dual-Rail Pre-charge Circuit with Binary Decision Diagram Architecture

In this section, we propose a novel DPA countermeasure to reduce the inevitable leakage at logic level, called Dual-rail Pre-charge circuit with Binary Decision Diagram architecture (DP-BDD). It is based on BDD and constructed in the following steps.

Pre-charged AND-OR gates. We avoid the existence of glitches to control the transition probability of all signals in a BDD circuit. In order to prevent glitches, we firstly replace 2-to-1 multiplexers in BDD to 2-way 2-and 4-input AND-OR (shortly, AND-OR) gates. As shown in Fig. 3, an AND-OR gate is equivalent to a 2-to-1 multiplexer except the negation of a select signal being input. Fig. 4(a) shows a modified BDD circuit. In the figure an isosceles trapezoid represents an AND-OR gate. Non-terminal nodes (A, \bar{A}) , (B, \bar{B}) , or (C, \bar{C}) are connected to each AND-OR gate as (sel, \overline{sel}) in Fig. 3.

Next, we apply so-called pre-charge mechanism to the terminal nodes $(0, 1)$ and the non-terminal nodes (A, \bar{A}) , (B, \bar{B}) , (C, \bar{C}) ; these signals are set to 0 on the pre-charge phase and evaluate to the corresponding value on the evaluation phase. We consider the output of an AND-OR gate at the lowest stage. On the evaluation

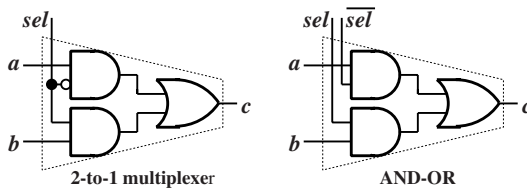


Fig. 3. A 2-to-1 multiplexer and an AND-OR gate

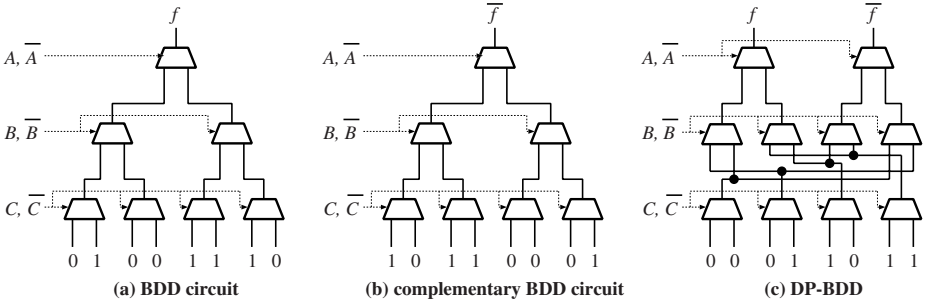


Fig. 4. Constructing DP-BDD

phase, all four inputs of an AND-OR gate perform either $(0 \rightarrow 0)$ or $(0 \rightarrow 1)$, then the output also performs either $(0 \rightarrow 0)$ or $(0 \rightarrow 1)$. On the pre-charge phase, all four inputs perform either $(0 \rightarrow 0)$ or $(1 \rightarrow 0)$, then the output also performs either $(0 \rightarrow 0)$ or $(1 \rightarrow 0)$. By adapting these transitions to the inputs of AND-OR gates at the next stage, we can confirm that all internal nodes and outputs of BDD have at most one transition both on the evaluation phase and on the pre-charge phase. Therefore, we can prevent glitches in the BDD circuit.

Appending complementary circuit. Preventing glitches doesn't give any guarantee to DPA resistance because the distribution of the transition activity depends on the inputs A, B, C . In order to make it independent of the inputs, we construct the complementary BDD circuit to the original BDD circuit. It can be simply created by exchanging 0 and 1 which are input to the terminal nodes as shown in Fig. 4(b). By appending the complementary circuit to the original circuit and merging them as shown in Fig. 4(c), one of the complementary AND-OR gates perform a transition both on the evaluation phase and on the pre-charge phase. Therefore, the activity factor within the merged circuit is constant independent of the input signals. We call such a merged circuit as Dual-rail Pre-charge circuit with Binary Decision Diagram architecture (DP-BDD) [1].

We consider the inevitable leakage, which is leakage caused by the difference of delay time between the input signals of complementary AND-OR gates shown in Fig. 5. We assume that all inputs of DP-BDD, non-terminal nodes and terminal nodes, are directly connected to registers and have no propagation delay except their setup time.

The difference of delay time between input signals of AND-OR gates may lead the difference of transition timing at the output which depends on some secret information. Since signals sel and \overline{sel} are directly connected to inputs of DP-BDD, the transition of sel and \overline{sel} occurs soon after the transition from the pre-charge phase to the evaluation phase, and the reverse transition. On the

¹ By inputting a random bit m and its negation \overline{m} to the terminal nodes instead of 0 and 1, all internal nodes and output of DP-BDD are easily masked by m . The addition of random masking, however, does not achieve secure design without special layout constraint according to the observation in [6][13].

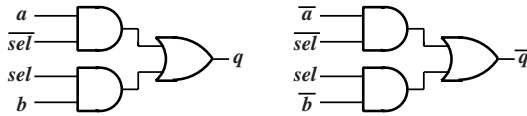


Fig. 5. Complementary AND-OR gates

pre-charge phase, the transition of q or \bar{q} occurs at the time when the transition of sel or \bar{sel} whether $sel = 0$ or 1 . On the evaluation phase, if $sel = 0$, the transition of the output signal q or \bar{q} occurs at the time when the transition of the input a or \bar{a} occurs; if $sel = 1$, the transition of q or \bar{q} occurs at the time when the transition of the input b or \bar{b} occurs. Therefore, the difference of delay time between a and b (or \bar{a} and \bar{b}) may leak the value sel on the evaluation phase. However, since the signals a and b (or \bar{a} and \bar{b}) pass the same number of AND-OR gates, the difference of delay time between these signals is relatively small, and then detecting the inevitable leakage by DPA is more difficult.

5 Application to AES S-Box

In order to protect hardware implementations of the Advanced Encryption Standard (AES) [10], the S-box is the most critical operation because it is the only non-linear operation in AES. In this section, we apply both WDDL and DP-BDD to implementations of AES S-box, and compare their effectiveness.

5.1 AES S-Box Based on WDDL (WDDL S-Box)

There are various ways to implement the AES S-box. The most compact implementation of AES S-box is that using composite fields [12,21,3]. We apply WDDL to the AES S-box described in [21], whose overall amount of gates is 103 XORs + 57 ANDs, because of its relatively short critical path.

Fig. 6 shows the schematic circuit of AES S-box using composite fields. There are several operations including an isomorphic mapping, multiplications and additions over Galois field. We notice path 1 and path 2 which both are the paths to the multiplication circuit over $\text{GF}(2^4)$. Path 1 has relatively short propagation delay because it passes only the isomorphic mapping circuit. On the other hand, path 2 has long propagation delay because it passes also the squaring, constant multiplication, addition, and inversion circuits over $\text{GF}(2^4)$ except the isomorphic mapping circuit. Thus, since the difference of delay time between path 1 and 2 are large, we guess the inevitable leakage caused by this difference can be detected by DPA.

5.2 AES S-Box Based on DP-BDD (DP-BDD S-Box)

Since the AES S-box has an 8-bit input and an 8-bit output, we firstly arrange eight binary decision trees of eight stages according to the truth tables of

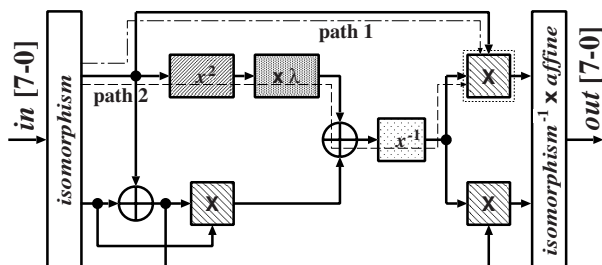


Fig. 6. AES S-box using composite fields

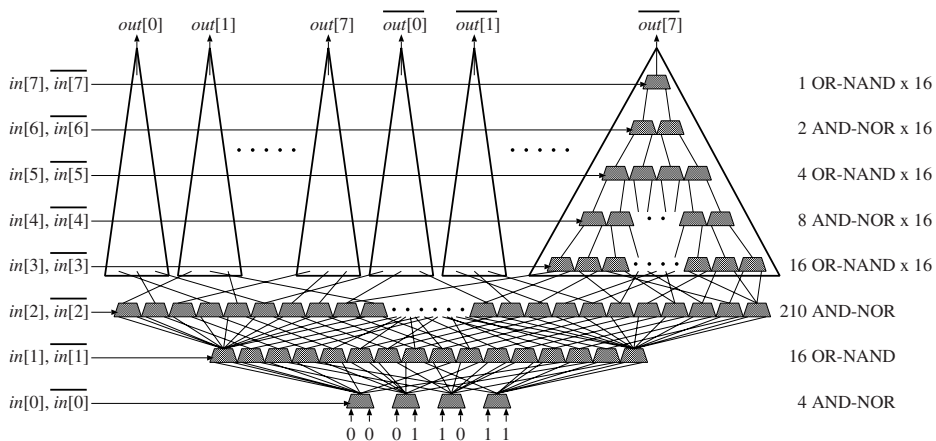


Fig. 7. AES S-box based on DP-BDD (DP-BDD S-box)

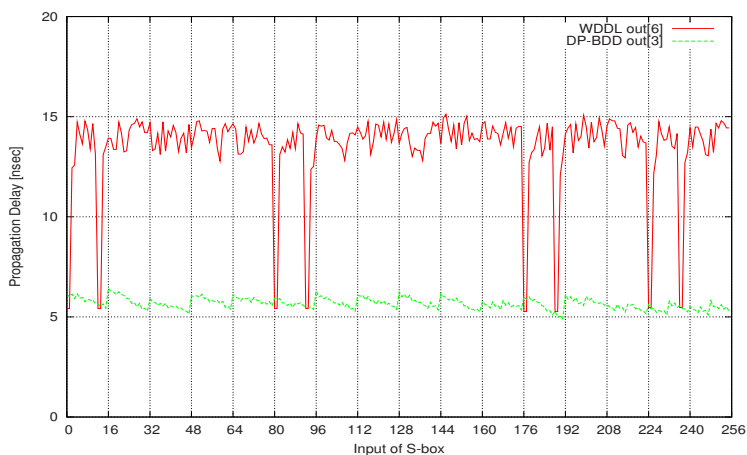


Fig. 8. Propagation delay of an output bit of WDDL S-box and DP-BDD S-box

AES S-box. Then, AES S-box based on DP-BDD (DP-BDD S-box) can be constructed in the way described in Section 4.

Fig. 7 shows the constructed DP-BDD S-box, where $in[i]$ denotes i -th bit of the input of the S-box and $out[i]$ denotes i -th bit of the output. In CMOS a positive gate is usually constructed out of a negative gate and an inverter, and then the use of positive gates is a disadvantage in terms of gate size. In order to reduce the gate size of DP-BDD S-box, we replace AND-OR gates to AND-NOR gates at the odd stages and to OR-NAND gates at the even stages, and then the input of OR-NAND gates are pre-charged to 1 on the pre-charge phase. Its overall amount of gates is 374 AND-NORs + 352 OR-NANDs. Since any path from the terminal node 0 and 1 to two input signals of an AND-NOR/OR-NAND gate passes the same number of AND-NOR/OR-NAND gates, the difference of delay time between the input signals of the gate is relatively small.

5.3 Experimental Results

We implemented both WDDL S-box and DP-BDD S-box, and performed netlist timing simulations to evaluate their effectiveness. The environment of our evaluation is as follows:

Language	Verilog-HDL
Design Library	0.18 μ m CMOS standard cell library
Simulator	VCS version 2006.06
Logic Synthesis	Design Compiler version 2006.06

One gate is equivalent to a 2-way NAND and the speed is evaluated under the worst-case conditions. In the library, an AND/OR gate, an AND-OR/OR-AND gate, and an AND-NOR/OR-NAND gate are equivalent to 5/4 gates, 9/4 gates, and 7/4 gates, respectively. These simulations are based on pre-routing delay, and then free from the incidental leakage caused by the automatization of the place-and-route.

We firstly evaluate the gate counts of WDDL S-box and DP-BDD S-box. An AND gate in the AES S-box is implemented using an AND gate and an OR gate in WDDL S-box as shown in Fig. 4, while an XOR gate in the AES S-box can be implemented using an AND-OR gate and an OR-AND gate. Thus the gate count of WDDL S-box is equivalent to $103 \times 9/2 + 57 \times 5/2 = 606$ excluding buffers. On the other hand, the gate count of DP-BDD S-box is equivalent to $374 \times 7/4 + 352 \times 7/4 = 1271$ excluding buffers.

Next, we evaluate the difference of transition timing at the output of logic gates in both WDDL S-box and DP-BDD S-box. Since we guessed the largest difference will occur at the output of the S-box, we searched the output bit of S-box that has the largest difference of transition timing for all possible 256 S-box inputs; $out[6]$ (or $out[6]$) and $out[3]$ (or $out[3]$) are the corresponding bits of WDDL S-box and DP-BDD S-box respectively. Fig. 8 shows the propagation delay of these bits for all 256 inputs; the above line shows that of WDDL S-box and the below line shows that of DP-BDD S-box. We confirmed that the maximum difference of transition timing at the output of DP-BDD S-box (1.526 ns) is about 1/6.5 of that of WDDL S-box (9.855 ns).

6 Towards Less Difference of Transition Timing

DP-BDD reduces the difference of transition timing at the output of AND-OR gates. It is, however, desirable to reduce this difference all the more since it could be detected by DPA. We consider that the difference occurs by the accumulation of the following factors:

- difference of propagation delay between input ports of each AND-OR gate,
- difference of load capacitance between input ports of each AND-OR gate,
- difference of the number of fan-out between output signals of AND-OR gates.

In order to reduce the influence of these factors, we apply delay adjustment to inputs of DP-BDD shown in Fig. 9.

On the pre-charge phase, we don't require any delay adjustment cell because the difference of transition timing at the output of each AND-OR gate is equivalent to the difference of propagation delay between input port of the AND-OR gate.

On the evaluation phase, we insert delay cells of $delay(a)$, $delay(b)$, and $delay(c)$ to (A, \bar{A}) , (B, \bar{B}) , and (C, \bar{C}) respectively. By inserting the delay cell of $delay(c)$ to (C, \bar{C}) , a transition of the output of AND-OR gates at stage 1 occurs at the time when a transition of C or \bar{C} reaches their input ports. Next, we set $delay(b)$ that satisfies $delay(b) - delay(c)$ is larger than the propagation delay from any input ports of AND-OR gates at stage 1 to any input ports of AND-OR gates at stage 2. That indicates that a transition of the output of AND-OR gates at stage 2 occurs at the time when a transition of B or \bar{B} reaches their input ports. Similarly, we set $delay(a)$ that satisfies $delay(a) - delay(b)$ is larger than the propagation delay from any input ports of AND-OR gates at stage 2 to any input ports of AND-OR gates at stage 3. Therefore, we can reduce the difference of transition timing at the outputs of all AND-OR gates to the difference of propagation delay between input port of the AND-OR gate also on the evaluation stage. It is very easy to satisfy these delay conditions because we have only to make the difference of delay between any two adjacent bits of the input sufficiently large.

By switching the input signals without delay and those with delay using AND gates, we can successfully reduce the difference of transition timing at all signals

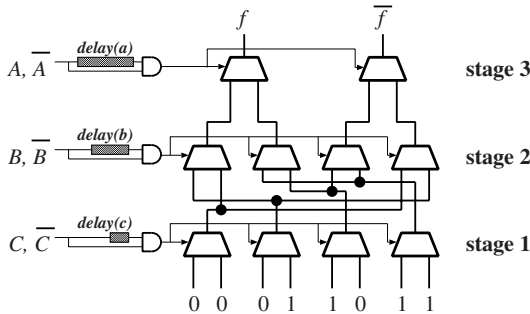


Fig. 9. Delay adjustment for DP-BDD

in DP-BDD in both the pre-charge stage and the evaluation stage. We confirmed that this delay adjustment reduced the maximum difference of transition timing in DP-BDD S-box to 0.018 ns (about 1/85 of that without delay adjustment), which is just the difference of propagation delay between the input ports \overline{sel} and sel of an OR-NAND gate.

7 Conclusion

In this paper we presented the logic-level DPA countermeasure called DP-BDD. DP-BDD has a dual-rail logic style and can be implemented using CMOS standard cell libraries. Our experimental results showed that DP-BDD can significantly reduce the difference of transition timing at the outputs of AES S-box compared to WDDL. We consider that DP-BDD is a practical and effective DPA countermeasure for implementations of S-boxes.

At CHES 2006, Homma et al. presented high-resolution waveform matching based on a Phase-Only Correlation (POC) techniques and its application to DPA [5]. They claimed that the POC-based techniques can evaluate the displacement between signal waveforms with higher resolution than the sampling resolution. One of further works we need to carry out is how large difference of the delay time between the input signals leads to DPA leakage in real devices using such techniques.

References

1. Akers, S.B.: Binary Decision Diagram. *IEEE Trans. on Computers* C-27(6), 509–516 (1978)
2. Bryant, R.E.: Graph-Based Algorithm for Boolean Function Manipulation. *IEEE Trans. on Computers* C-35(8), 677–691 (1986)
3. Canright, D.: A Very Compact S-Box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
4. Chen, Z., Zhou, Y.: Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side Channel Leakage. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 242–254. Springer, Heidelberg (2006)
5. Homma, N., Nagashima, S., Imai, Y., Aoki, T., Satoh, A.: High-Resolution Side-Channel Attack Using Phase-Based Waveform Matching. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 187–200. Springer, Heidelberg (2006)
6. Gierlichs, B.: DPA-Resistance Without Routing Constraints? In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 107–120. Springer, Heidelberg (2007)
7. Guilley, S., Hoogvorst, P., Mathieu, Y., Pacalet, R.: The “Backend Duplication” Method. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 383–397. Springer, Heidelberg (2005)
8. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
9. Mangard, S., Popp, T., Gammel, B.M.: Side-Channel Leakage of Masked CMOS Gates. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg (2005)

10. National Institute of Standard and Technology (NIST), Advanced Encryption Standard (AES). FIPS Publication 197 (2001)
11. Popp, T., Mangard, S.: Masked Dual-Rail Pre-Charge Logic: DPA-Resistant without Routing Constraints. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 172–186. Springer, Heidelberg (2005)
12. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A Compact Rijndael Hardware Architecture with S-box Optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
13. Schaumont, P., Tiri, K.: Masking and Dual-Rail Logic Don't Add Up. In: Paillier, P., Verbaauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 95–106. Springer, Heidelberg (2007)
14. Suzuki, D., Saeki, M., Ichikawa, T.: DPA Leakage Models for CMOS Logic Circuits. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 366–382. Springer, Heidelberg (2005)
15. Suzuki, D., Saeki, M.: Security Evaluations of DPA Countermeasures Using Dual-Rail Pre-Charge Logic Style. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 255–269. Springer, Heidelberg (2006)
16. Suzuki, D., Saeki, M., Ichikawa, T.: Random Switching Logic: A New Countermeasure against DPA and Second-Order DPA at the Logic Level. IEICE Transactions 90-A(1), 160–168 (2007)
17. Tiri, K., Akmal, M., Verbaauwhede, I.: A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In: ESSCIRC 2002, pp. 403–406 (2002)
18. Tiri, K., Verbaauwhede, I.: A Logic Level Design Methodology for A Secure DPA Resistant ASIC or FPGA Implementation. In: DATE 2004, pp. 246–251 (2004)
19. Tiri, K., Verbaauwhede, I.: Place and Route for Secure Standard Cell Design. In: CARDIS 2004, pp. 143–158 (2004)
20. Trichina, E.: Combinational Logic Design for AES SubByte Transformation on Masked Data. IACR Cryptology ePrint Archive 2003 /236 (2003), <http://eprint.iacr.org/2003/236>
21. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC Implementation of the AES S-boxes. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 67–78. Springer, Heidelberg (2002)
22. Yang, C., Ciesielski, M., Singhel, V.: BDS: A BDD Based Logic Optimization System. In: Proc. of the 37th ACM/IEEE DAC 2000, pp. 92–97 (2000)

SCARE of an Unknown Hardware Feistel Implementation

Denis Réal^{1,2}, Vivien Dubois¹, Anne-Marie Guilloux¹,
Frédéric Valette¹, and Mhamed Drissi²

¹ CELAR, 35 Bruz, France

{Denis.Real,Vivien.Dubois:Anne-Marie.Guilloux,
Frederic.Valette}@dga.defense.gouv.fr

² INSA-IETR, 20 avenue des coesmes, 35043 Rennes, France

{Denis.Real,Mhamed.Drissi}@insa-rennes.fr

Abstract. Physical attacks based on Side Channel Analysis (SCA) or on Fault Analysis (FA) target a secret usually manipulated by a public algorithm. SCA can also be used for Reverse Engineering (SCARE) against the software implementation of a private algorithm. In this paper, we claim that an unknown Feistel scheme with an hardware design can be recovered with a chosen plaintexts SCA attack. First, we show that whatever is the input of the unknown Feistel function, its one-round output can be guessed by SCA. Using this relation, two attacks for recovering the algorithm are proposed : an expensive interpolation attack on a generic Feistel scheme and an improved attack on a specific but commonly used scheme. Then, a countermeasure is proposed.

1 Introduction

Cryptographic algorithms are designed for being robust against logical analysis. However, the activity of any given device (smart cards, FPGA, microprocessor...) filter through side channels such as the processing time, the power consumption or the electromagnetic radiations. Secret values are then vulnerable to statistical attacks such as Differential Power Analysis (DPA) [4] or Correlation Power Analysis (CPA) [1], but also to one-shot attacks such as Template Attacks (TA) [2] [8].

Side Channel Analysis For Reverse Engineering (SCARE) techniques are also employed for recovering a private algorithm. R. Novak proposed a strategy for identifying a substitution table of a secret implementation of the GSM A3/A8 algorithm [6]. The feasibility of a SCARE for symmetric cryptography have been proved by recovering a software DES implementation [3]. Indeed, the intermediate results on one round are available by SCA: they are computed sequentially due to the software design. In this article, we go further with a SCARE attack against an hardware implementation of a generic Feistel Scheme. The SCA feasibility is demonstrated on a hardware DES implemented on an ASIC . However, the side channel leakage exploitation is done for a generic Feistel Scheme. Then, we propose a countermeasure against this generic SCARE attack.

2 The Hardware Implementation of a Generic Feistel Scheme

2.1 Assumption on the Feistel Scheme Design

The logic function is an expensive resource on a cryptographic device, especially on a smart card. As a consequence, the Feistel function is usually implemented once and the input registers are updated at each round. For example, the right part of the first round output is xored with the left part of the plaintext and is assumed to be written on a register whose value was the right part of the plaintext. This realistic design assumption is detailed on Fig. 1.

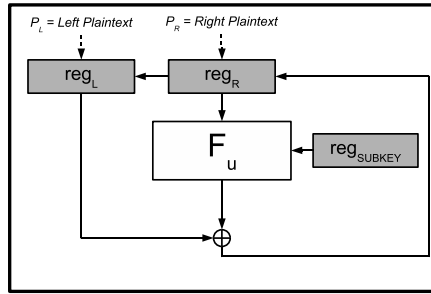


Fig. 1. Assumption on a Generic Feistel Scheme Design

2.2 The Power Consumption During a Bit Transition

The power consumption of a D Flip-Flop (DFF) depends on its current activity. Due to CMOS transistor behavior, register state changes make the DFF consume power [9]. When the logic value 1 is applied a DFF input while the previous value was 0, a current rise occurs on the rising edge of the clock. Furthermore, the current rise during this transition is higher than the static state holding dissipation. Then, we approximate the circuit behavior by saying that it supplies energy for bit transition and does not supply energy for holding a bit value. Lines supplying energy to the circuit can be observed with electromagnetic near field techniques. The effect of bits transitions occurring at the same instant are also built up on the side channel electromagnetic radiations. This is the power supplying line leakage model. Even if it is experimentally harder, an attacker can also observe the current at the DFF level with the same kind of techniques [5]. The radiations at the DFF level do not behave the same than previously. Then a transition from 0 to 1 induces a radiation opposite to the radiation due to a transition from 1 to 0 due to current motions in the DFF. This is transistor current leakage model. The choice of the leakage model depends on the part of the circuit targeted by the probe. State of the art electromagnetic techniques permits to target power supplying line: the DFF level is harder to get. The SCA we propose matches with the power supplying line leakage model.

3 Side Channel Analysis of a Generic Feistel Scheme

3.1 Side Channel Identification of a Feistel Scheme

SCA techniques permit to identify a Feistel scheme. Indeed, the right part of the plaintext is used once on the 1st round as R_0 round but also on the 2nd round as L_1 . Then, a CPA mean computed on the right part of the plaintext contains two peaks (one on the 1st round and the other on the 2nd round). A CPA computed on the left part of the plaintext contains one peak one on the 1st round. Fig. 2 illustrates the effect on a DES hardly implemented. The electromagnetic radiations are measured using near field techniques and a probe sensitive to the vertical magnetic field. Fig. 2.1 shows one ciphering operation. Fig. 2.2 is a CPA on the right part of the plaintext and Fig. 2.3 is a CPA on the left part of the plaintext.

A Feistel Scheme can then be identified with two DPA traces. This method for characterizing a generalized Feistel Scheme can also be used with ciphertexts.

3.2 SCA for to Guess the One Round Output of the Feistel Function

Our SCA is a chosen plaintext attack. We assume that the attacker is able to spy N ciphering operations. Let P_i be the corresponding plaintexts with L_i and R_i their corresponding left and right hand halves, $i < N$. The right part of the plaintext is chosen fixed to R : $R_i = R$. Let R_i^j and L_i^j be the input of the j^{th}

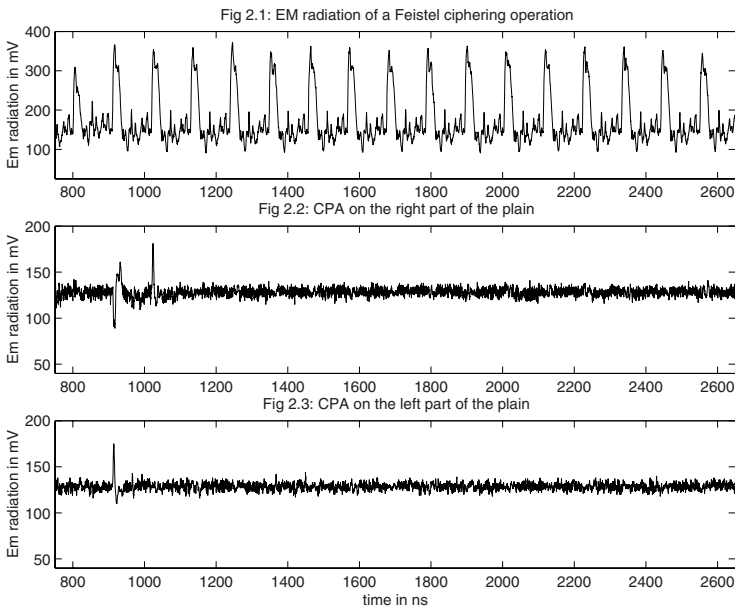


Fig. 2. CPAs on the plaintext

- N : The number of plaintexts.
- P_i : The i^{th} plaintext with $i < N$.
 - L_i : The corresponding left part.
 - $R_i = R$: The corresponding fixed right part .
- F_u : The unknown Feistel Function.
- L_i^j : The left input of the j^{th} round : $L_i^0 = L_i$.
- R_i^j : The right input of the j^{th} round : $R_i^0 = R$.
- Y_i^j : The output of F_u at the j^{th} round.
- Y : The fixed output of F_u after the round 0 $Y = F_u(R)$.
- X : The CPA object, classification according to the hamming weight of $L_i \oplus X$.

Fig. 3. Notations

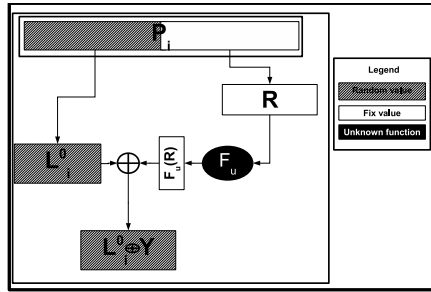


Fig. 4. SCA context

round of the feistel scheme : $R_i^0 = R_i = R$, $L_i^0 = L_i$ and $L_i^1 = R_i$. Let F_u be the unknown Feistel function and Y_i^j its corresponding output for the j^{th} round : $Y_i^j = F_u(R_i^j)$. Let Y be the fixed output of F_u for the round 0: $Y_i^0 = F_u(R) = Y$ and $R_i^1 = L_i \oplus Y$. The notations for this chosen plaintexts context are reminded in Fig. 3 and Fig. 4

We consider 4-bits words: the leakage of 4-bits can usually be properly observed while an exhaustive search on 4 bits in not very expensive. This choice corresponding to the S-box size of the DES is purely coincidental. Let $L_{i,0}$, X_0 and R_0 be the first 4 bits of L_i , R and X . Here, we propose an SCA for guessing X_0 , the generalization to the other words of X being trivial. For each possible value α of X_0 , we compute the side channel trace Γ_α being the mean trace computed using the plaintexts P_i whose first word is α : we obtain then 16 traces. Let X_0 be the index of the trace Γ_α which is minimal at the $t = T_0 + T_d$. We claim that $X_0 = Y_0 \oplus R_0$. Indeed, according to the assumption made on the Feistel design scheme, we clearly see that, at the instant $t = T_0 + T_d$ the value $L_i \oplus Y$ overwrites the value R on the register 2. The value of L_i that minimizes the number of bit transition occurring on the register 2 is $L_i = Y \oplus R$. The classification done computing the traces Γ_α is random regarding all the 4-bits words of L except the first one explaining we are able to isolate its leakage from

the logical noise. Then we can guess the value of $Y : Y = X \oplus R$. This SCA attack permit to have the relation $Y = F_u(R)$ with X and Y known and F_u the unknown feistel function. Fig. 5 shows the results with $R_0 = 0 \times 3$ and $Y_0 = 0 \times B$ and $N = 5000$. Fig. 5.1 illustrates a ciphering operation useful for finding the instant $t = T_0 + T_d$. Fig. 5.2 shows the mean traces $\Gamma_{0 \times 8}$ and $\Gamma_{0 \times 7}$: $\Gamma_{0 \times 8}$ is the minimum at $t = T_0 + T_d$ while, as expected $\Gamma_{0 \times 7}$ is the maximum ($7 \oplus 8 = 0 \times F$). Fig. 5.3 shows the 14 other mean trace Γ_α . The difference before the instant $t = T_0 + T_d$ cannot be used because the signal is not time-stationary.

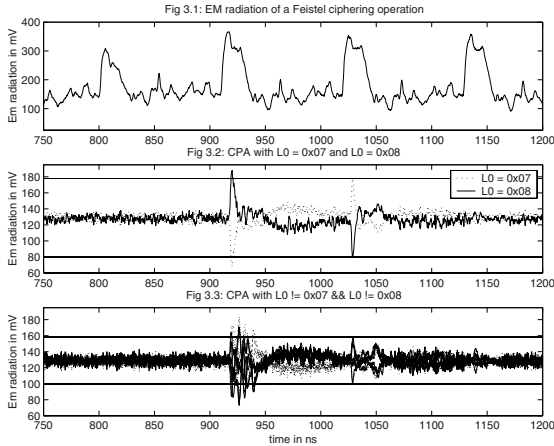


Fig. 5. Γ_α mean traces

Then, a low cost SCA permits to compute the output of the unknown Feistel function F_u of any input word R . This SCA aims at being general. In the case of measurements with a low level of noise as is observed here, the attacker can search $F_u(R)$ bit by bit. Indeed, he can compare two side channel traces linked to two plaintexts whose difference is one bit of the left half : the correct bit value corresponds to the lower radiation. Then the attacker can expect to get $F_u(R)$ with 64 chosen plaintexts for DES.

4 The Cryptographic Attack Derived from the SCA

In this section, we compute the number of (chosen) input-output pairs required to fully recover the unknown function F_u . We first compute this number for a general function F_u of given degree and input/output size. Secondly, we show that this number can be made much smaller when F_u follows a specific but commonly used design.

4.1 Simple Interpolation Attack

F_u is a vectorial function with n input/output bits. Its bitwise coordinates are polynomials of the n input bits of degree denoted d . For each input-output pair,

we use the input and the j -th bit of the output to reconstruct the j -th coordinate of F_u . When the same input-output pairs can be used to reconstruct the individual coordinates of F_u , the number of input-output pairs that we need can be computed from the reconstruction of a single coordinate of F_u . Since each such polynomial has about $n^d/d!$ coefficients, we need this number of input-output pairs to be able to resolve the unknown coefficients by linear algebra. The computational cost of recovering one coordinate of F_u is therefore $n^{3d}/(d!)^3$ binary operations and n times this number to recover F_u entirely. For $n = 64$ and $d = 8$, this amounts to roughly 2^{32} input-output pairs and 2^{102} binary operations.

4.2 Improved Attack on a Class of Commonly Used Schemes

The specification of a general function F_u requires a large amount of storage for practical choices of degree and input size. For this reason, functions admitting a compact description are often considered in that place. A commonly encountered class of functions is built by composition of a random function of a reduced number of bits $d \ll n$ with a linear compression from n bits to d bits. In this case, each of the n coordinates of F_u has the shape $f \circ S$ where f is an unknown function from d bits to 1 bit and S is an unknown linear function from n bits to d bits. We next show how to recover f and S from a few input-output pairs of $f \circ S$.

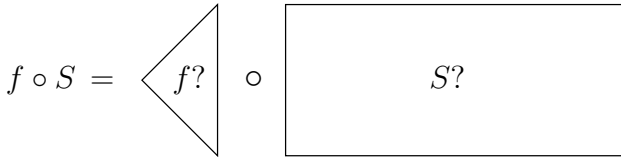


Fig. 6. A representation of $f \circ S$

First, we observe that considering a composed function $f \circ S$ the problem of recovering f and S has multiple solutions. Indeed, for any linear permutation ϕ of the rows of S , the transformed $f \circ \phi^{-1}$ and $\phi \circ S$ are an alternative solution. As a consequence, when the first d columns of S are linearly independent, we can suppose that they actually form the Identity matrix. Then, considering the inputs of $f \circ S$ with their last $n - d$ bits to 0 actually provides us with input-output pairs of f . There are 2^d inputs of $f \circ S$ with their last $n - d$ bits to 0 and this is what we need to compute f by the interpolation method. Finally, the cost of recovering f is 2^d input-output pairs of $f \circ S$ and 2^{3d} binary operations.

We are now left at finding the coefficients of the $n - d$ last columns of S . For this, we sequentially resolve the coefficients of the k -th column for $k = d+1, \dots, n$ by using inputs of $f \circ S$ with their k -th bit to 1 and their last $n - k$ bits to 0. At each step, the coefficients of the previous columns are known and each input-output pairs of $f \circ S$ with the prescribed bits to 0 defines an algebraic relation in the d unknown coefficients of the current column. Since we want to keep to a minimum the number of input-output pairs of $f \circ S$ that we use, we do not use

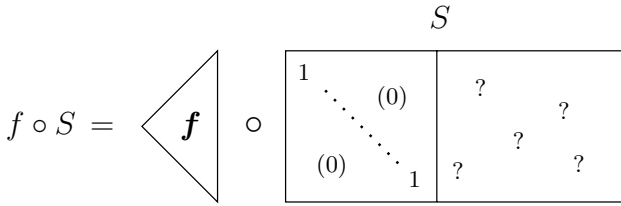


Fig. 7. Updated representation of $f \circ S$

these algebraic relations to solve the coefficients of the current column. Instead, we do exhaustive search on these coefficients and use the algebraic relations to identify their correct values. Since each algebraic relation is satisfied with probability about 1/2, we need on average about d input-output pairs of $f \circ S$ to discriminate the correct values of all d coefficients. Applying this method to the $n - d$ unknown columns of S costs $(n - d)d$ input-output pairs of $f \circ S$ and $(n - d)2^d$ binary operations.

In all, our improved method allows to recover the complete description of $f \circ S$ from $2^d + (n - d)d$ chosen input-output pairs and using $2^{3d} + (n - d)2^d$ binary operations. For $n = 64$ and $d = 8$, this amounts to roughly $2^{9.5}$ input-output pairs and 2^{24} binary operations.

5 Proposition of Countermeasure

Hardware countermeasures are either expensive to design, either not efficient against statistical SCA [7]. We propose then the logic countermeasures based on the attacker capability.

5.1 Countermeasure with the Supplying Line Leakage Model

The current rise during the transitions from 0 to 1 or 1 to 0 is much higher than the static dissipation observed for the holding of state. The proposed SCA is based on the knowledge of the first value of a register. But, if the targeted register is precharged with a random before the secret-dependent value be written, the proposed SCA cannot be done. That countermeasure implementation costs one register more but also costs a random generator and specific means to load it in the register. However, random generator are usually implemented on smart cards. Fig. 8 illustrated that the precharging countermeasure design. During the initialization stage, the plaintext is written in a register while the random value is loaded in the other one. At the first round of the algorithm, right part of the plaintext is processed but the result is now written on the randomized register. A random value is needed at the 1^{st} round, the cryptographic algorithm making this register precharged value be random for the other rounds. This design divides the data rate by 2. As the 2 first and 2 last round of the algorithm are sensitive to our way of attacking, the designer can improve the data rate by precharging only these round which costs 4 clock periods.

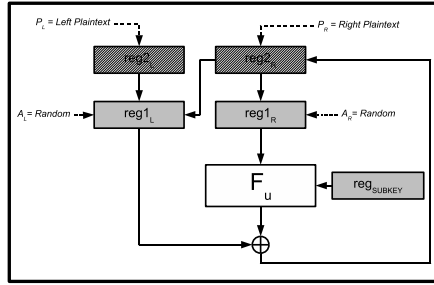


Fig. 8. Precharging countermeasure

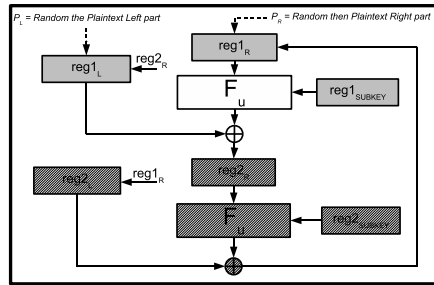


Fig. 9. Unrolling implementation

Fig. 9 presents another way of designing that countermeasure. The Feistel function is unrolled on two round. The initialization consists then on loading a random value on the rising clock edge and the plaintext on the next rising clock edge. The data rate is not divided by 2 anymore, just 1 clock period is added. However, this design multiplies the logic by 2. With this design, two more logic had been implemented, the cost is then area and consumption not time execution anymore.

Precharging is a well-known countermeasure. However it seems to be inefficient in the case of DFF leakage model.

5.2 Improved Countermeasure in the Case of DFF Leakage Model

If the falling or the rising transition can be distinguished, the proposed SCA is still possible, at the difference that the attacker does not search the minimal mean radiation but the medium mean radiation. This SCA bypasses the previous countermeasure. Indeed, the outputs on one round of the Feistel could be divided into three groups and instead of two groups. Then, at the bit level, an attacker who observe no transition cannot say anything. But a transition from 0 to 1 means that the observed bit of $L \oplus F_u(R)$ is 1 and a transition from 1 to 0 means that the observed bit of $L \oplus F_u(R)$ is 0. Then, the proposed attack work with the register precharging, it just costs more experiments. The previous countermeasure is then improved by working randomly with the plaintext or its complementary. Due to Feistel properties, the final result can be calculated

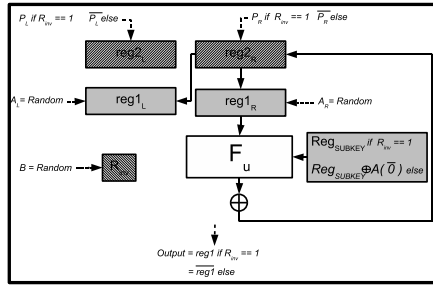


Fig. 10. Complementary Precharging

whatever be the kind of input data. Then the attacker who observes a 0 in a register does not know if this bit is inverted or not. This countermeasure resists to 1^{st} order SCA. If F_u can be expressed as $F_u(R) = F_2(K \oplus A(R))$ with A an affine function, the countermeasure implementation can be improved as represented on Dig. 10.

6 Conclusion

State of art in SCARE attack usually target software implementation. The attack we presented in this article is, to our knowledge, the first attack for reverse engineering a general Feistel with an hardware design. All the results proposed here can be applied to a generalized Feistel Scheme. We presented also an improved countermeasure mixing complementary register and precharging. This countermeasure should resist to 1^{st} order SCA. It has not been design for to resist to 2^{nd} order SCA. However, for that a 2^{nd} order SCA attack be a real threat, its feasibility should be studied in future works and its complexity clearly identified.

References

1. Brier, É., Clavier, C., Olivier, F.: Correlation power analysis with a kleakage model. In: Joyeand, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
2. Chari, S., Rao, J.R., Rihotgi, P.: Template Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
3. Daudigny, R., Ledig, H., Muller, F., Valette, F.: SCARE of the DES (side channel analysis for reverse engineering of the data encryption standart). In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 393–406. Springer, Heidelberg (2005)
4. Kocher, T., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
5. Mangard, S., Popp, T., Gammel, B.M., Jun, B.: Side Channel Leakage of Masked CMOS Gates. In: Rao, J.R., Sunar, B. (eds.) CT-RSA 2005. LNCS, vol. 3376, pp. 361–365. Springer, Heidelberg (2005)

6. Novak, R.: Side-channel attack on substitution blocks. In: Zhou, J., Yung, M., Han, Y. (eds.) ACNS 2003. LNCS, vol. 2846, pp. 307–318. Springer, Heidelberg (2003)
7. Réal, D., Canovas, C., Clédière, J., Drissi, M., Valette, F.: Defeating Classical Hardware Countermeasures: a New Processing for Side Channel Analysis. In: Desing Automation Test in Europe International conference – DATE 2008 (2008)
8. Rechberger, C., Oswald, E.: Practical Template Attacks. In: Lim, C.H., Yung, M. (eds.) WISA 2004. LNCS, vol. 3325, pp. 440–456. Springer, Heidelberg (2005)
9. Sze, S.M.: Semiconductor Devices: Physics and Technology. John Wiley and Sons, Inc, Chichester (2002)

Evaluation of Java Card Performance

Samia Bouzefrane¹, Julien Cordry¹,
Hervé Meunier², and Pierre Paradinas³

¹ CNAM 292 rue Saint-Martin 75003 Paris France
`firstname.lastname@cnam.fr`

² INRIA POPS Parc Scientifique de la Haute Borne Bt. IRCICA 50,
avenue Halley - BP 70478 59658 Villeneuve d'Ascq, France
`herve.meunier@lifl.fr`

³ INRIA Rocquencourt 78150 Le Chesnay France
`Pierre.Paradin@inria.fr`

Abstract. With the growing acceptance of the Java Card standard, understanding the performance behaviour of these platforms is becoming crucial. To meet this need, we present in this paper, a benchmark framework that enables performance evaluation at the bytecode and API levels. We also show, how we assign, from the measurements, a global mark to characterise the efficiency of a given Java Card platform, and to determine its performance according to distinct smart card profiles.

Keywords: Java Card, Benchmark, Performance, Test.

1 Introduction

The advent of the Java Card standard has been a major turning point in smart card technology. It provides a secure, vendor-independent, ubiquitous Java platform for smart cards. It shortens the time-to-market and enables programmers to develop smart card applications for a wide variety of vendors' products.

In this context, understanding the performance behaviour of Java Card platforms is important to the Java Card community. Currently, there is no solution on the market which makes it possible to evaluate the performance of a smart card that implements Java Card technology. In fact, the programs which realize this type of evaluations are generally proprietary and not made available to the whole of the Java Card community. Hence, the only existing and published benchmarks are used within research laboratories (e.g., SCCB project from CEDRIC laboratory [3,6], or IBM Research [1]). However, benchmarks are important in the smart card area because they contribute in discriminating companies products, especially when the products are standardised.

Our purpose is to describe the different steps necessary to measure the performance of the Java Card platforms. In this paper, the emphasis is towards determining the optimal parameters to enable measurements that are as accurate and linear as possible. We also show, how we assign, from the measurements, a global mark to characterise the efficiency of a given Java Card platform, and to determine its performance according to distinct smart card profiles.

The remainder of this paper is organised as follows. In Section 2, we describe the Java Card technology. Subsequently, we detail in Section 3 the different modules that compose the framework architecture. Section 4 presents a state of the art of the benchmarking attempts in smart card area before concluding the paper in Section 5.

2 Java Card and Benchmarking

2.1 Java Card Technology

Java Card technology provides means of programming smart cards [2,8] with a subset of the Java programming language. Today's smart cards are small computers, providing 8, 16 or 32 bits CPU with clock speeds ranging from 5 up to 40MHz, ROM memory between 32 and 128KB, EEPROM memory (writable, persistent) between 16 and 64KB and RAM memory (writable, non-persistent) between 3 and 5KB. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816-4 standard). The communication is done in master-slave mode. It is always the terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). In the case of Java powered smart cards, the cards ROM contains, in addition to the operating system, a Java Card Virtual Machine (JCVM), which implements a subset of the Java programming language, hence allowing Java Card applets to run on the card.

A Java Card applet should implement the `install` method responsible for initializing the applet (usually called by the applet constructor) and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host. More than one applet can be installed on a single card, however only one can be active at a time (the active one is the most recently selected by the Java Card Runtime Environment – JCRE). A normal Java compiler is used to convert the source code into Java bytecodes. Then a converter must be used to convert the bytecode into a more condensed form (CAP format) that can be loaded onto a smart card. The converter also checks that no unsupported features (like floats, strings, etc.) are used in the bytecode. This is sometimes called off-card or off-line bytecode verification.

2.2 Addressed Issues

Our research work falls under the MESURE project [12], a project funded by the French administration (ANR¹), which aims at developing a set of open source tools to measure the performance of Java Card platforms.

Only features related to the normal use phase of Java Card applications will be considered here. Are excluded features like installing, personalizing or deleting

¹ <http://www.agence-nationale-recherche.fr/>

an application since they are of lesser importance from user's point of view and performed once.

Hence, the benchmark framework enables performance evaluation at three levels:

- The VM level: to measure the execution time of the various instructions of the virtual machine (basic instructions), as well as subjacent mechanisms of the virtual machine (e.g., reading and writing the memory).
- The API level: to evaluate the functioning of the services proposed by the libraries available in the embedded system (various methods of the API, namely those of Java Card and GlobalPlatform).
- The JCRE level: to evaluate the non-functional services, such as the transaction management, the method invocation in the applets, etc.

The set of tests are supplied to benchmark Java Card platforms available for anybody and supported by any card reader. The various tests thus have to return accurate results, even if they are not executed on precision readers. We reach this goal by removing the potential card reader weakness (in terms of delay, variance and predictability) and by controlling the noise generated by measurement equipment (the card reader and the workstation). Removing the noise added to a specific measurement can be done with the computation of an average value extracted from multiple samples. As a consequence, it is important on the one hand to perform each test several times and to use basic statistical calculations to filter the trustworthy results. On the other hand, it is necessary to execute several times in each test the operation to be measured in order to fix a minimal duration for the tests (> 1 second) and to expect getting precise results.

We will not take care of features like the I/Os or the power consumption because their measurability raises some problems such as:

- For a given smart card, distinct card readers may provide different I/Os measurements.
- Each part of an APDU is managed differently on a smart card reader. The 5 bytes header is read first, and the following data can be transmitted in several way: 1 acknowledge for each byte or not, delay or not before noticing the status word.
- The smart card driver used by the workstation generally induces more delay on the measurement than the smart card reader itself.

3 General Benchmarking Framework

3.1 Introduction

We defined a set of modules as part of the benchmarking framework. The general framework is illustrated in the figure [□](#).

The benchmarks have been developed under the Eclipse environment based on JDK 1.6, with JSR268. The underlying ISO 7816 smart card architecture forces us

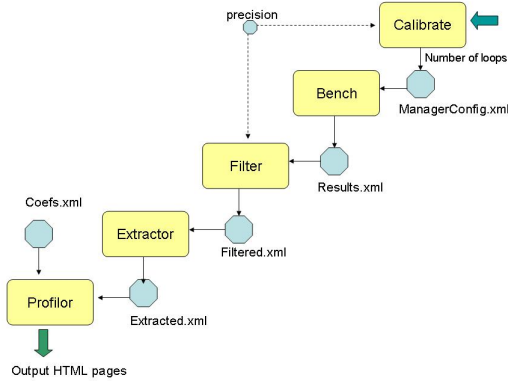


Fig. 1. Overall Architecture

to measure the time a Java Card platform takes to answer to a command APDU, and to use that measure to deduce the execution time of some operations.

The benchmarking development tool covers two parts: the script part and the applet part. The script part, entirely written in Java, defines an abstract class that is used as a template to derive test cases characterized by relevant measuring parameters such as, the operation type to measure, the number of loops, etc. A method `run()` is executed in each script to interact with the corresponding test case within the applet. Similarly, on the card is defined an abstract class that defines three methods:

- a method `setUp()` to perform any memory allocation needed during the lifetime test case.
- a method `run()` used to launch the tests corresponding to the test case of interest, and
- a method `cleanUp()` used after the test is done to perform any clean-up.

The testing applet is capable of recognizing all the test cases and launching a particular test by executing its `run` method.

Our Eclipse environment integrates the Converter tool from Sun Microsystems, which is used to convert a standard Java Card applet class into a JCA file during a first step. This file is completed pseudo-automatically by integrating the operations to be tested with the Java Card Assembly instructions, as we explain in the following paragraph. The second step consists in capgenerating the JCA file into a CAP file, so that the applet could be installed on any Java Card platform.

3.2 Modules

In this section, we describe the general benchmark framework that has been designed to achieve the MESURE objective. The methodology consists of different

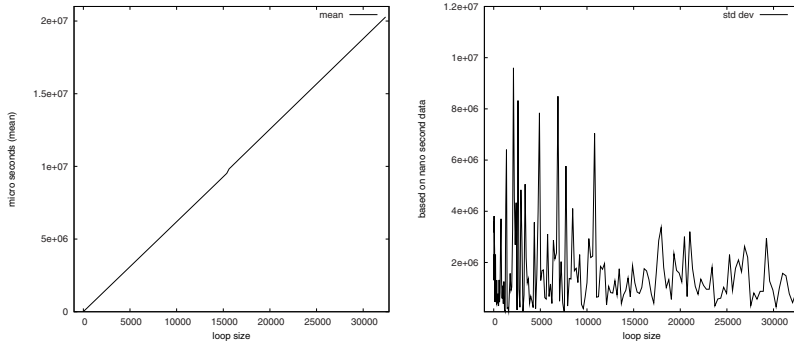


Fig. 2. Raw measurement and standard deviation

steps. The objective of the first step is to find the optimal parameters used to carry out correctly the tests. The tests cover the VM operations and the API methods. The obtained results are filtered by eliminating non-relevant measurements and values are isolated by drawing aside measurement noise. A profiler module is used to assign a mark to each benchmark type, hence allowing us to establish a performance index for each smart card profile used. In the following subsections, we detail every module composing the framework.

The bulk of the benchmark consists in performing time execution measurements while we send APDUs from the computer through the Card Acceptance Device (CAD) to the card. Each test (*run*) is performed a certain number of times (Y) to ensure reliability of the collected execution times, and within each *run* method, we perform on the card a certain number of loops (L). L is coded on the byte P_2 of the APDUs which are sent to the on-card applications. The size of the loop performed on the card is $L = (P_2)^2$.

The Calibrate Module. The calibrate module computes the optimal parameters (such as the number of loops) needed to obtain measurements of a given precision.

Benchmarking the various different bytecodes and API entries takes time. At the same time, it is necessary to be precise enough when it comes to measuring those execution times. Furthermore, the end user of such a benchmark should be allowed to focus on a few key elements with a higher degree of precision. It is therefore necessary to devise a tool that let us decide what are the most appropriate parameters for the measurement.

Figure 2 depicts the evolution of the raw measurement, as well as its standard deviation, as we take 30 measurements for each available loop size of a test applet. As we can see, the measured execution time of an applet grows linearly with the number of loops being performed on the card (L). On the other hand, the perceived standard deviation on the different measurements varies randomly as the loop size increases, though with less and less peaks. Since a bigger loop size means a relatively more stable standard deviation, we can use both the

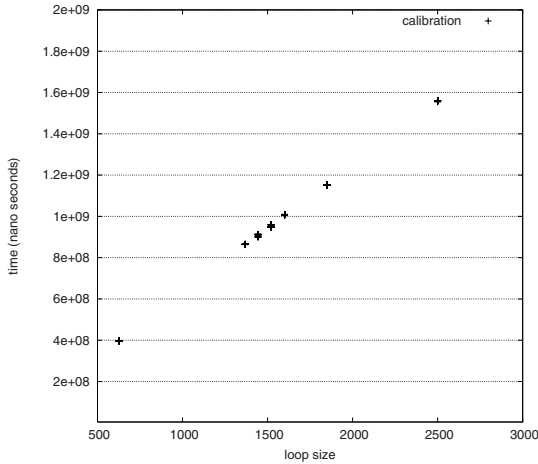


Fig. 3. A sample calibration

standard deviation and the mean measured execution time as a basis to assess the precision of the measurement as follows.

To assess the reliability of the measurements, we compare the value of the measurement with the standard deviation. The end user will need to specify this ratio between the average measurement and the standard deviation, as well as an optional minimum accepted value, which is set at one second by default.

With both the ratio and the minimal accepted value, as specified by the end user, we can test and try different values for the loop size to binary search and approach the ideal value. In figure 3, we try to calibrate a test by first trying out a loop size of 2500. The program decided that the set of 30 obtained values was too precise and therefore too time demanding. It then tried to evaluate the precision of the test for a loop size of 625. Since the measurements were below the minimum value, the program then tried to perform the same evaluation for a loop size of 1369, and so on, until we reached a loop size for which both conditions were satisfied.

The Bench Module. For a number of cycles, defined by the calibrate module, the bench module performs the measurements for:

- The VM byte codes
- The API methods
- The JCRE mechanisms (such as transactions)

The development of some of the test applets is detailed in [18].

The Filter Module. Experimental errors lead to noise in the raw measurement experiments. This noise leads to imprecision in the measured values, making it difficult to interpret the results. In the smart card context, the noise is due to crossing the platform, the CAD and the terminal (measurement tools, OS, hardware).

The issues become: how to interpret the varying values and how to compare platforms when there is some noise in the results. The filter module uses a statistical design to extract meaningful information from noisy data. From multiple measurements for a given operation, the filter module uses the mean value μ of the set of measurements to guess the actual value, and the standard deviation σ of the measurements to quantify the spread of the measurements around the mean. Moreover, since the measurements respect the normal Gaussian distribution (see figure 4), a confidence interval $[\mu - (n \times \sigma), \mu + (n \times \sigma)]$, within which the confidence level is of $1 - a$, is used to help eliminate the measurements outside the confidence interval, where n and a are respectively the number of measurements and the temporal precision, and they are related by traditional statistical laws.

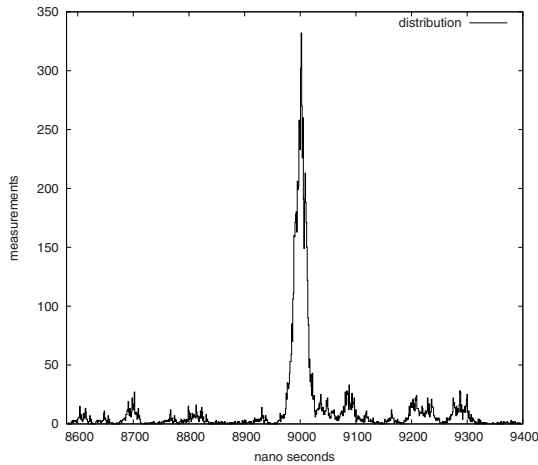


Fig. 4. The distribution of 10000 measured execution time

The Extractor Module. The extractor module is used to isolate the execution time of the features of interest among the mass of raw measurements that we gathered so far.

Benchmarking bytecodes and API methods within Java Card platforms requires some subtle means in order to obtain execution results that reflect as accurately as possible the actual isolated execution time of the feature of interest. This is because there exists a significant and non-predictable elapse of time between the beginning of the measure, characterized by the starting of the timer on the computer, and the actual execution of the bytecode of interest. This is also the case the other way around. Indeed, when performing a request on the card, the execution call has to travel several software and hardware layers down to the card's hardware and up to the card's VM (vice versa upon response). This non-predictability is mainly dependent on hardware characteristics of the benchmark environment (such as the card acceptance device (CAD), PC's hardware, etc), the OS level interferences, services and also on the PC's VM.

To minimize the effect of these interferences, we need to isolate the execution time of the features of interest, while ensuring that their execution time is sufficiently important to be measurable.

The maximization of the bytecodes execution time requires a test applet structure with a loop having a large upper bound, which will execute the bytecodes for a substantial amount of time. On the other hand, to achieve execution time isolation, we need to compute the isolated execution time of any auxiliary bytecode upon which the bytecode of interest is dependent. For example if `sadd` is the bytecode of interest, then the bytecodes that need to be executed prior to its execution are those in charge of loading its operands onto the stack, like two `sspush`. Thereafter we subtract the execution time of an empty loop and the execution time of the auxiliary bytecodes from that of the bytecode of interest to obtain the isolated execution time of the bytecode. As presented in figure 5, the actual test is performed within a method (`run`) to ensure that the stack is freed after each invocation, thus guaranteeing memory availability.

Applet framework	Test Case
<pre>process() { i = 0 While i <= L DO { run() i = i+1 } }</pre>	<pre>run() { op₀ op₁ ⋮ op_{n-1} op_n }</pre>

Fig. 5. Test framework for a bytecode op_0

In figure 5:

- L represents the chosen loop upper bound;
- op_n represents the operation of interest;
- op_i for $i \in [0..n - 1]$ represents the auxiliary bytecodes necessary to perform the operation op_n .

To compute the mean isolated execution time of op_n we need to solve a system with the following equations:

$$\overline{M(op_n)} = \frac{\overline{m_L(op_n)} - \overline{m_L(Emptyloop)}}{L} - \sum_{i=0}^{n-1} \overline{M(op_i)}$$

Where:

- $\overline{M(op_i)}$ is the mean isolated execution time of the operation op_i .
- $\overline{m_L(op_i)}$ is the mean global execution time of the operation op_i , including interferences coming from other operations performed during the measurement, both on the card and on the computer, with respect to a loop size L .

These other operations represent for example auxiliary bytecodes needed to execute the operation of interest, or OS and JVM specific operations. The mean is computed over a significant number of tests. It is the only value that is experimentally measured.

- *Emptyloop* represents the execution of a case where the `run` method does nothing.

The formula presented above implies that prior to computing $\overline{M(op_n)}$ we need to compute $M(op_i)$ for $i \in [0..n - 1]$. The system can be solved as long as the dependency relation between the operations is well founded, and that there is a set of operations that do not depend on any other operation.

The Profiler Module. In order to define performance references, our framework provides measurements that are specifically adapted to one of the following application domains:

- banking applications
- transport applications
- identity applications.

A Java Card VM is instrumented in order to count the different operations performed during the execution of a script for a given application. More precisely, this virtual machine is a simulated and proprietary VM executing on a workstation. This instrumentation method is rather simple to implement compared to a static analysis based methods, and can reach a good level of precision, but it requires a detailed knowledge of the applications and of the most significant scripts.

Some features related to bytecodes and API methods appeared to be necessary and the simulator was instrumented to give useful information such as:

- for the API methods:
 - the types and values of method parameters
 - the length of arrays passed as parameters,
- for the bytecodes:
 - the type and duration of arrays for array related bytecodes (load, astore, arraylength),
 - the transaction status when invoking the bytecode.

A simple utility tool has been developed to parse the log files generated by the instrumented Java Card VM, which builds a human-readable tree of method invocations and bytecode usage.

Thus, with the data obtained from the instrumented VM, we attribute for each application domain a number that represents the performance of some representative applets of the domain on the tested card. Each of these numbers is then used to compute a global performance mark.

We use weighted means for each domain dependent mark. Those weights are computed by monitoring how much each Java Card feature is used within a regular use of standard applets for a given domain. For instance, if we want to

test the card for a use in transport applications, we will use the statistics that we gathered with a set of representative transport applets to evaluate the impact of each feature of the card.

We are considering the measurement of the feature f on a card c for an application domain d . For a set of n_M extracted measurements $M_{c,f}^1, \dots, M_{c,f}^{n_M}$ considered as significant for the feature f , we can determine a mean $\overline{M_{c,f}}$ modeling the performance of the platform for this feature.

Given n_C cards for which the feature f was measured, it is necessary to determine the reference mean execution time R_f , which will then serve as a basis of comparison for all subsequent tests.

Hence the “mark” $N_{c,f}$ of a card c for a feature f , is the relation between R_f and $\overline{M_{c,f}}$:

$$N_{c,f} = \frac{R_f}{\overline{M_{c,f}}}$$

However, this mark is not weighted. For each pair of a feature f and an application domain d , we associate a coefficient $\alpha_{f,d}$, which models the importance of f in d . The more a feature is used within typical applications of the domain, the bigger the coefficient:

$$\alpha_{f,d} = \frac{\beta_{f,d}}{\sum_{i=1}^{n_F} \beta_{i,d}}$$

where:

- $\beta_{f,d}$ is the total number of occurrences of the feature f in typical applications of the domain d .
- n_F is the total number of features involved in the test.

Therefore, the coefficient $\alpha_{f,d}$ represents the occurrence proportion of the feature of interest f among all the features.

Hence, given a feature f , a card c and a domain d , the “weighted mark” $W_{c,f,d}$ is computed as follows:

$$W_{c,f,d} = N_{c,f} \times \alpha_{f,d}$$

The “global mark” $P_{c,d}$ for a card c and for a domain d is then the sum of all weighted marks for the card. A general domain independant note for a card is computed as the mean of all the domain dependant marks.

3.3 Unused Features

The document [19] details the included and the excluded features. Only features related to the normal use phase of Java Card applications are considered here. Measuring the performance when installing, personalizing or deleting an application, is of less importance from the user’s point of view. Moreover, these management operations are only performed once. As a consequence, the constructors of the Java Card API, as well as methods such as `Applet.register()`,

etc. are not measured. Besides, we focus on success paths and not on the failure ones, on account of their relevance. Then, failure cases such as the comparison methods of the Java Card API `equals(...)` on a bad AID (`OwnerPIN.check(...)`) on a bad PIN ...), as well as `Exception` classes are not taken into account. In the same respect, some bytecodes, that are never used in a regular application are not measured here.

4 State of the Art

Currently, there is no standard benchmark suite which can be used to demonstrate the use of the JCVm and to provide metrics for comparing Java Card platforms. In fact, even if numerous benchmarks have been developed surrounding the JVM (see [3]), there are few works that attempt to evaluate the performances of smart cards. The first interesting initiative has been done by Jordi et al. in [17] where they study the performance of micro-payment for Java Card platforms, i.e., without PKI. Even if they consider Java Card platforms from distinct manufacturers, their tests are not complete as they involve mainly computing some hash functions on a given input, including the I/O operations. A more recent and complete work has been undertaken by Erdmann in [15]. This work mentions different application domains, and makes the distinction between I/O, cryptographic functions, JCRE and energy consumption. Infineon Technologies is the only provider of the tested cards for the different application domains. The software itself is not available. The work of Fischer in [16] compares the performance results given by a Java Card applet with the results of the equivalent native application. Another interesting work is that carried out by the IBM BlueZ secure systems group and concretized through a Master thesis [11]. JCOP framework has been used to perform a series of tests to cover the communication overhead, DES performance and reading and writing operations into the card's memory (RAM and EEPROM). Markantonakis in [9] presents some performance comparisons between the two most widely used terminal APIs, namely PC/SC and OCF. Papapanagiotou et al. in [10] evaluate the performance of two on-line certificate revocation and validation protocols on two different Java Card platforms in order to determine which protocol is more efficient for smart card use. Chaumette et al. in [13,14] show the performance of a Java Card grid with respect to the scalability of the grid and with different types of cards.

5 Conclusion

In this paper, we have proposed a methodology aiming at characterizing the performance of Java Card platforms by measuring different levels of benchmarks using measurement techniques to analyze the platform's performance. This work was undertaken as part of a project funded by the French administration MESURE. The Java Card Benchmarking framework is now accessible on-line (see [12]) since it is published as an open-source tool. Our work focuses on

measuring the execution time of the virtual machine bytecodes, the API methods and the JCRE mechanisms.

All the measured features are based on the Java Card 2.2 platforms. With the publication of the Java Card 3.0 specifications [20], two versions are proposed. While the Connected Edition features a new virtual web-oriented machine, the Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and targets more resource-constrained devices that support traditional applet-based applications. Hence, the majority of the features measured in Measure tool will be reused in this edition. However, all the new features such as those based on 32-bit integers are not considered.

Currently, we are working on the prediction of the execution time of the applications, by using formal methods.

References

1. Cap, C.H., Maibaum, N., Heyden, L.: Extending the Data Storage Capabilities of a Java-based Smart card. In: Sixth IEEE Symposium on Computers and Communications (ISCC 2001). IEEE, Los Alamitos (2001)
2. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Reading (2000)
3. Douin, J.-M., Paradinas, P., Pradel, C.: Open Benchmark for Java Card Technology, e-Smart Conference (September 2004)
4. GemXpresso Reference Manual, Gemplus (1998)
5. Sm@rtCafe Reference Manual Giesecke & Devrient (1999)
6. Grimaud, G., Paradinas, P., Vétillard, E.: Measuring the performance of the Java Card Platform, Java One (May 2006)
7. Guyot, V., Boukhatem, N., Pujolle, G.: Smart Card performances to handle Session Mobility. ICI, IFIP/IEEE (September 2005)
8. Java Card 2.2.2 Specification (April 2006), <http://java.sun.com/products/javacard/>
9. Markantonakis, C.: Is the performance of smart card cryptographic functions the real bottleneck? In: 16th international conference on Information security: Trusted information: the new decade challenge, vol. 193, pp. 77–91. Kluwer, Dordrecht (2001)
10. Papapanagioutou, K., Markantonakis, C., Zhang, Q., Sirett, W.G., Mayes, K.: On the Performance of Certificate Revocation Protocols Based on a Java Card Certificate Client Implementation. In: 20th IFIP International Information Security Conference (Sec 2005) - Small Systems Security and Smart cards (May 2005)
11. Rehioui, K.: Java Card Performance Test Framework, Université de Nice, Sophia-Antipolis, IBM Research internship (September 2005)
12. The MESURE project MESURE, <http://mesure.gforge.inria.fr/Eng/Index>
13. Chaumette, S., Grange, P., Karray, A., Sauveron, D., Vignéras, P.: Secure distributed computing on a Java Card Grid, LaBRI, Université Bordeaux 1, 1331-04, (2004), <http://www.labri.fr/publications/paradis/2004/CGKSV04>
14. Atallah, E., Darrigade, F., Chaumette, S., Karray, A., Sauveron, D.: A Grid of Java Cards to Deal with Security Demanding Application Domains. In: 6th edition e-Smart conference & demos, Sophia Antipolis, French Riviera (September 2005), <http://www.labri.fr/publications/paradis/2005/ADCKS05>

15. Erdmann, M.: Benchmarking von Java Card, LudwigMaximilians-Universität München, Institut für Informatik (May 2004)
16. Fischer, M.: Vergleich von Java und Native-Chipkarten Toolchains, Benchmarking, Messumgebung, LudwigMaximilians-Universität München, Institut für Informatik (2006)
17. Castellà-Roca, J., Domingo-Ferrer, J., Herrera-Joancomartí, J., Planes, J.: A Performance Comparison of Java Cards for Micropayment Implementation. In: CARDIS, pp. 19–38 (2000)
18. Paradinas, P., Cordry, J., Bouzefrane, S.: Performance Evaluation of Java Card Bytecodes WISTP, pp. 127–137 (May 2007)
19. Functionalities of the MESURE tools : <http://mesure.gforge.inria.fr/pub/documents/F2.1Functionalities1.0.pdf>
20. Java Card 3.0, <http://java.sun.com/javacard/3.0/>

Application of Network Smart Cards to Citizens Identification Systems

Joaquin Torres¹, Mildrey Carbonell¹, Jesus Tellez², and Jose M. Sierra¹

¹ Carlos III University of Madrid, Computer Science Department,
Avda. de la Universidad, 30, 28911, Leganés (Madrid), Spain
{jtmarque, mcarbhone, sierra}@inf.uc3m.es

² University of Carabobo, Venezuela, Computer Science Department (Facyt)
Av. Universidad, Sector Bárbula, Valencia, Venezuela
jtellez@uc.edu.ve

Abstract. This paper proposes a new authentication and authorization architecture based on a *network smart card* with identification purposes: ID-NSCard. Thus, a citizen who holds this kind of device might be securely authenticated by a remote authoritative server in an identification system. This work shows how the standardized specifications are transparently reused and integrated in the proposed architecture. Details of the protocol and authentication mechanisms are provided for a Case of Study: Spanish National Electronic ID Card.

1 Introduction

Multiple works have attempted to define what a person's identity is. Many of them consider identity as the distinguishing characteristics that determine unequivocally that a person is who that person claims to be. The authentication is the mechanism by which the identity of a person is verified.

Identity clearly is a target of theft. By stealing another person's identity, somebody could gain access to services or facilities to which the thief is not entitled. Stronger ways of reinforcing security and trust are needed in order to avoid undesirable impersonations.

Many countries are starting to issue national identity cards or electronic passports for citizens that include a chip card (ID-card). This is an electronic way to hold a trusted identity credential. Additionally, both logical and biometric identifiers are usually required for authenticating the citizens' credentials during the identification process.

Most of these ID-card solutions provide two main security services: authentication and digital signature. Note that the first one allows the authoritative organisation to determine whether the claimed identity really belongs to the service requester (in this context, identification and authentication terms are commonly used), and the second one guarantees the non-repudiation of an electronic transaction. With these goals, the law [1] and standardization bodies [2, 3] envisage two different qualified digital certificates: citizen's authentication certificate and citizen's digital signature certificate, both of them installed in a SSCD (secure signature creation device).

The present paper is just focused on the identification scheme, by means of the authentication of identity credentials. More concretely, it is focused on a remote

authentication procedure, which does not take place between a smart card (in a SSCD role) and an access terminal, but between the first one and a remote authentication server, where the authoritative application is running.

One of the more relevant European references for the implementation and deployment of national ID cards are issued by the European Committee for Standardization (or CEN), which among other specifications is defining an application interface for smart cards used as secured signature creation devices [4, 5] and the European Citizen Card, ECC [6]. Nevertheless, the protocols and schemes derived from these standards might be improved in terms of robustness and security. After analyzing them, we propose a more secure Identification System based on ID-NSCards, which aim to be more autonomous smart cards with identification purposes. With these goals, an atomic implementation of layer 2 authentication protocols within the card and end-to-end communications with a back-end authentication/authorization server, among other aspects, are presented in this work.

In the reminder of this paper, the related work is reviewed and analyzed in section 2 and, afterwards, we describe an authentication architecture based on our network smart card concept, NSCard, which implements ID-card authentication functionalities (ID-NSCard). In section 4, security and trust issues related to such an architecture are discussed. Finally, we treat a case of study for developing the proposed architecture in a real identification system: the Spanish National electronic ID-card.

2 Related Works

One of the objectives of our work is, as far as possible, to treat the smart card as a networked host. Several works have been done in this area.

The proposal in [8] was oriented towards establishing a simplified TCP/IP protocol stack. The smart card supports this protocol stack and behaves like a small Web server. The (U)SIM security modules were not absent from this approach. New perspective on business models favoured the creation of generic tools based on smart cards, such as SIM Toolkit [9]. In this frame, these devices are equipped with a certain level of pro-activity and an improved connectivity, through a client-server model, in an over-the-air (OTA) system. This technology is based on Short Message Service, SMS, and is able to update SIM cards as well as downloading and activating new services. Interesting research was done making use of this technology, enabling the SIM card to be used as Web server [10]. Its implementation does not correspond entirely to that established in the standard HTTP protocol, but the result is functional and effective for certain applications.

The aim in [11] was to obtain a TCP-type protocol. This protocol did not fulfil all of the requirements that are established in the standard [12] but it included the concept of agent-based Internet card. In [13] Internet infrastructure extends to include smart cards for the first time, and a specific middleware is defined in order to protect communications between applications and smart cards. A proxy implementation was noted in [14-16]. This allowed cards to be efficiently integrated in distributed environments. The consolidation of Java Card as object-oriented programming favoured this process. Java Card Web Servlet technology was used to transform the smart card into a portable repository for Web objects, including HTML pages with data for a

specific application, in [17]. Smart cards continue being integrated into the Web environment, even merely as an element with the capacity to store and transport user's personal information securely. The use of these devices to manage Web session cookies was proposed in [18].

In [19] an important number of experts discussed the characteristics, steps and planning of what could be a new generation of smart cards with or without contacts. A clear evolution towards a networked smart card was quite evident. More recent works specify what these cards would be like and what security advantages they would introduce. These cards could fall under the ever-widening concept of network smart cards. More details on the essential contributions of these works could be found in the following paragraphs.

In [20] and [21], the card was already treated as an Internet node which implemented standardised security and communication protocols, to be connected to a network via the host. The card was able to provide services or access Internet resources making use of protocol stacks in the same way as any other node on the network. Its use in security solutions was soon proposed. In this way, the network smart card was able to establish secure direct communication with remote Internet servers, as shown in [22]. This capacity allows the cards to guarantee online transactions. An authentication comparison with OTP devices or with regard to conventional cards can be found in [23]. Other works were focused on lower-level security and they treated packet filtering by the smart card in different stages. This filtering may be produced from interruptions service routines to the actual filtering by the protocol stack [24].

More recently, the advances in networks smart cards have been studied in [25, 26] from different approaches. Concretely in [25], we describe the rationale of our network smart card concept.

Regarding the use of network smart cards in remote identification schemes based on ID-cards there is not much work done. In this paper, we aim to securely integrate this kind of devices in such schemes with a reduced protocol stack and guaranteeing security and trust features.

2.1 Analysis of the Closer ID-Card Solution for User Authentication and Authorization

Many of the current European ID-card solutions for citizen authentication are based on the standard [5] (e.g. German Electronic ID card, Finnish eID card, Spanish National Electronic ID card, etc.). This specification states the common scenario where the citizen identification remotely takes place: it is named "Client/Server Authentication". This procedure is described in Figure 1 for case of the SSL protocol. Briefly, the citizen's authentication certificate (client certificate) is used during the SSL handshake and afterwards such a citizen is challenged with the value T. For completing the tunnel establishment, she signs T and replies. Obviously, the signature is securely computed inside the ID-card. Once the authentication server verifies the signature, the citizen is authenticated (identified) and the secure tunnel is finally established between the user's computer and the remote server. Consequently, some on-line services will be then available.

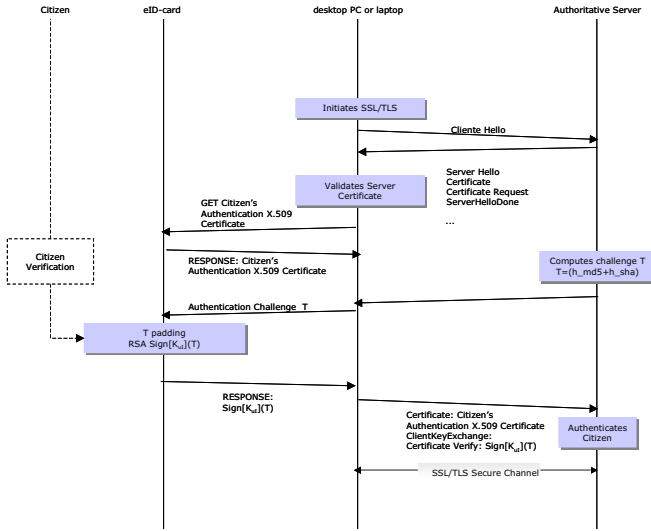


Fig. 1. Client/Server Authentication derived from [5]

But the reader should note in this scheme that:

- It is not applicable to any remote identification scenario: an equipment with a complete TCP/IP protocol stack and secure socket layer is needed. In such a scenario, this equipment is a desktop PC or laptop.
- The secure tunnel is established between PC and remote sever. In a public (unknown) environment, an end-to-end tunnel between the ID-card and the remote server should be desirable.
- Any device authentication does not take place. The ID-card is not explicitly authenticated. In [4], it is specified that a mutual device authentication shall be used if the operating environment of the ID-card cannot be entirely trusted (untrustworthy environment). This may be the case in public signature terminals or other devices that cannot provide a trusted channel. In the case of requiring a remote authentication process, a *device authentication* should be performed. After successful mutual device authentication, session keys are available on both sides to be used in subsequent transmissions. The appropriate secure messaging should be in compliance with ISO/IEC 7816-4 [7].

Taking the previous constraints into account, our work aims to design a complete authentication and authorization architecture for an identification system, which represents a more robust and flexible solution in terms of security, with the following features:

- The authentication protocol will be implemented as an integral part of the ID-card, with the goal of isolating the protocol of the implementation in the access terminal (e.g. laptop, desktop PC, PDA, etc.). Therefore, our approach considers an ID-card with autonomy during the authentication process. In other words, the ID-card participates as stand-alone supplicant or claimant,

and not relies on the access terminal (i.e. equipment or host providing the card reader) for this functionality.

- Layer 2 authentication based on a network smart card, NSCard, [25]: we propose the ID-card integration in a layer 2 authentication scheme, which is based on EAP protocol. Therefore, a lightweight networking protocol stack is easily supported by the smart card (TCP/IP and upper layers are not required). We define in this paper an EAP-ID method, which refers to a generic authentication method with identification purposes on our architecture.
- End-to-end mutual authentication scheme: the ID-card and the remote authentication server participate as tunnel endpoints. The individual identification is securely performed through such a tunnel.

Additionally, this work assumes an *a priori* untrustworthy environment, where the access terminal is considered as a potential attacker. Therefore, a previous mutual device authentication has been defined in our identification scheme.

In the following section of this paper, a new proposal of an authentication and authorization architecture along with a network smart card, with specific identification purposes (ID-NSCard), are defined.

3 A New Identification System Based on ID-NSCards

This paper proposes a new authentication architecture for ID-cards systems based on our network smart card concept. Under this scope, we consider a remote authentication and authorization scheme, where the ID-card adopts the functionality of stand-alone supplicant instead of split supplicant ("split supplicant" means that ID-card and the access terminal (hereafter referred as Access Control Equipment, ACE) cooperate in the authentication process as an unique device). That is why, in our work, the authentication protocol stack is designed as an integral part of the ID-card (atomic design). With this goal, we propose a specific protocol stack for the chip card that participates as actual end in the authentication process with a remote AAA server. This protocol stack is illustrated in Figure 2. The upper layer EAP-ID represents a generic EAP authentication method [27], specifically here designed with individuals identification purposes. Therefore, the EAP-ID method handles the credentials associated to the duplet individual-domain and the related cryptographic algorithms, during the identification process. Usually, most of the robust identification schemes require a password/PIN for controlling a private key, which is associated to a public key authentication certificate (e.g. X.509v3 certificates) and additionally they may require a biometric token.



Fig. 2. The protocol stack in the ID-NSCard

Note that in our approach, the goal of a generic EAP-ID method is not to add a new authentication protocol or method but adapt existing authentication protocols used in previous standardized identification schemes. This protocol stack is here defined with a general purpose. Hence, in this section we refer to an "individual", considering that she could be both an user/member registered with an organization and a citizen in a governmental domain. In section 5, the implementation of the authentication method in Spanish National Electronic Identity Card (named DNI-e) as an example of EAP-ID method for citizens is profusely described.

A complete end-to-end architecture is represented in Figure 3. This new architecture introduces significant advantages and requires minimal changes in the network side. Thus AAA proxies keep settings and implementation features. Regarding the Access Control Equipment, ACE, a simple implementation of standardized protocols allows it to behave as access point (or NAS, Network Access Server) to the network with pass-through authenticator functionalities. Hence, this equipment must implement EAP and RADIUS client protocols according to [28]. For simplicity's sake, we refer to RADIUS protocol in this paper, but note that a more robust protocol such as DIAMETER [29] could be also implemented in our architecture.

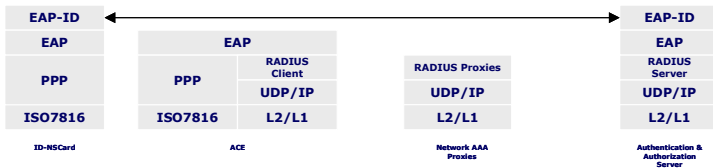


Fig. 3. Authentication and Authorization protocol architecture based on our ID-NSCard

In a first step, the RADIUS server authenticates the Access Control Equipment (ACE) by their own mechanisms. After this step, the functionality of the pass-through authenticator is already shifted to ACE. This reinforces the stand-alone supplicant functionality in the ID-card, since ACE cannot act as supplicant and authenticator at the same time for the same ID-card. One should note the advantages that the ID-card isolation brings with regard to assure the security of the entire scheme in untrust-worthy scenarios. More security and trust issues are discussed in section 4 of this paper.

Our architecture takes advantage of the functions of the LCP protocol provided by PPP [30]. LCP/PPP protocol may be easily accommodated in the ID-card stack. The functions for controlling network included in the NCP sub-protocol are beyond the scope of this work. On the other hand, PPP offers versatility in authentication, thanks to its extensibility. In fact, EAP (Extensible Authentication Protocol) was initially designed for PPP. According to our approach, the EAP Layer must be implemented atomically in the smart card and must allow for exchanging of packets between the EAP methods and LCP frames, as well as, for controlling duplicates and retransmissions.

The EAP-ID method should be designed with the goal of security reinforcing. Hence, three phases should be considered in the authentication process. The first one is regarding the mutual authentication between RADIUS entities. The ACE and AAA server proceed with a previous establishment of shared secret keys and mutual

authentication process. After this phase, the ACE is allowed to perform access control functionalities as an authenticator (pass-through) in the EAP scheme and the communication between ACE and the remote server will be protected. Hence, the second phase should be oriented to establish a end-to-end secure tunnel between the actual ID-NSCard and the authentication remote server. Obviously, such a tunnel establishment should take place after a mutual device authentication. That means that both devices (ID-NSCard and server) must posse their own authentication tokens (and independently on user credentials). Basically, two key mechanisms could be applied in this step: shared secret keys or public key certificates. The latter requires the usage of card verifiable certificates (according to ISO 7816). In this work, we describe a common end-to-end tunnel establishment, which uses shared secret keys. A generic EAP-ID must be able to perform the following protocol:

Assume that ID-NSCard (C) and authentication server(S) know the 3DES encryption key k_{ENC} , and the MAC computation key, k_{MAC}

$C \rightarrow S$: $SN_C || RND_C$, the 8-bytes serial number SN_C unequivocally associated to C and a fresh 8-bytes random number RND_C .

$S \rightarrow C$: authentication cryptogram $ACG1$, as function among others of SN_C and RND_C

C: verifies $ACG1$ (S is authenticated), generates the send sequence counter SSC_C and derives the session key K_{SK}

$C \rightarrow S$: authentication cryptogram $ACG2$

S: verifies $ACG2$ (C is authenticated), generates the corresponding SSC_S , also derives the session key K_{SK}

$C \leftrightarrow S$: further communication is protected by a secure channel (K_{SK} encryption)

Once the mutual device authentication is successful and the secure channel is established, the individual (cardholder) is required to be identified by means of her associated identity credentials in the third phase. Therefore, the procedure through the tunnel continues as follows:

Assume that a X.509v3 public key certificate ($Certificate_1$) and the corresponding private key K_{rI} with authentication purposes have been issued for an individual I.

$S \rightarrow C$: challenge T

I: cardholder is required for entering the corresponding password to sign

$C \rightarrow S$: $Sign[K_{rI}](T) || Certificate_1$

S: verifies the digital signature, $Verify[K_{ul}](Sign[K_{rI}](T))$, and the identity of individual I is authenticated by server S.

Derived from this protocol and our architecture (Figure 3), an authentication message exchange has been designed in our work. An example is described in section 5 of this paper.

4 Notes about the Testbed

A testbed of our authentication and authorization architecture for a identification system has been developed. The back-end authentication server is basically implemented in a computer where freeRADIUS [31] is running, which provides API support both EAP/RADIUS and EAP methods development. Additionally, it implements a set of state machines of EAP (Extensible Authentication Protocol), for an EAP backend authenticator. The EAP API is extended in order to support EAP-ID as a new authentication method including the corresponding method state machine and message parsing. On the other hand, the OpenSSL library includes a general purpose cryptography library, which is partially included in this testbed with the goal of providing well-known cryptographic functionalities.

Multiple network AAA proxies could intermediate between the ACE and the authoritative server. Our testbed considers just one proxy, which simulates one of these entities. The standard RADIUS protocol procedure in a relay version allows us to complete the implementation of the adequate protocol stack in an IEEE 802.11 wireless access point. The Access Control Equipment, ACE, is implemented by a common laptop with an IEEE 802.11g wireless interface. The functionality of RADIUS in this equipment is performed by JRadius-Client [32], a Java version of a NAS Client. The technical challenges for rolling out commercial Access Control Equipments with these features have been easily responded. Note that many of the current PDAs and smart phones with ISO-7816 interfaces are programmable. This protocol functionality that is proposed in our work could be transparently implemented as a library (e.g. dll dynamic library) for the OS. Therefore, the impact in existing terminals is minimized and a potential large-scale deployment is clearly feasible.

The bulk LCP/EAP protocol stack -according to the standardized state machines- has been implemented in a G&D Sm@rtCafé Expert 3.x smart card and it has been enhanced with the corresponding EAP-ID method functionalities.

5 Security and Trust Model Discussions

Regarding security aspects of our architecture, it should be noted that we are not proposing a completely new authentication protocol in the context of identification systems. Our architecture is designed by well-known protocols that are implemented inside the ID-card with a novel approach.

Nevertheless, this new architecture determines a new way to transport authentication messages between the ID-NSCard and the authentication and authorization server, and where the ID-NSCard takes the control in the user side. Therefore, the security weakness and threats are derived by the actual nature of such standardized protocols and the correctness of their implementation.

Additionally, new secure algorithms, key material or cryptographic techniques are not required. The implementation of the algorithms and authentication mechanisms is transparently reused [4, 5], in both sides. However, one of the more important impacts of our proposal is related to the trust models. If we study the trust model derived from the current scenario detailed in Figure 1, we observe that there exists an explicit trust between the PC and the authentication server (supported by SSL protocol). In any

case, the trust relationship in the interface between access terminal (i.e. PC) and ID-card is not questioned and it could be considered as "blind". As we mentioned before, this assumption should not be applied to all scenarios and a more flexible solution is required. With this goal, we have introduced a more robust architecture, which a new trust model is derived from. Therefore, it could be adapted to multiple wired/wireless scenarios, even in mobility situations.

In our trust model, the trust relationship between the access terminal (ACE) and the authentication server is supported and protected by RADIUS protocol and such a trust relationship should be considered as explicit. Here, the ACE is part of the network and it behaves as an access point for the ID-NSCard. The trust relationship between ID-NSCard and ACE should be a priori null (untrustworthy). After an end-to-end successful authentication process (supported by an EAP-ID method) between the ID-NSCard and the authentication server, the trust relationship between them should be then considered explicit, since it is a mutual device authentication process. Therefore, in this step the trust relationship between ID-NSCard and ACE is implicit, since any direct mutual authentication process between them has not occurred. In other words, *iff* ID-NSCard trusts authentication server then the former trusts access terminal. This is a reasonable result in a priori untrustworthy scenarios.

After this step, the environment should be considered as trustworthy and just in these conditions the individual identification should be securely performed.

6 Application of the ID-NSCard to the Spanish Electronic ID Card

The application of the ID-NSCard to Spanish National Electronic ID card has been studied in our work. The Spanish National ID card aims to prove digitally the identity and other personal data of the owner by means of an authentication process and validating the integrity and signature of signed documents. Both goals are addressed by a chip card and two different public keys created inside. As result, a citizen X.509v3 authentication certificate and a citizen X.509v3 digital signature certificate are manageable by the owner. The policy requirements for implementing this identification system is based on [33, 34]. The security technical specifications are basically gathered from [4, 5]. Additionally, the envisaged services and scenarios where the Spanish National ID-card might be used are determined by the Police Authority, which depend on the Spanish Home Office. The Spanish case could be considered one of the pioneer experiences in EU.

Nevertheless, other potential scenarios could be considered. Suppose a police control (e.g. dangerous or critical transportation, highroads controls, border areas or government facilities, etc.) requires the identification of the individuals. Hence, the authoritative person (let's say a policeman) carries a reduced-size and portable equipment with a wireless interface (e.g. PDA with a chip card reader). This equipment implements the ACE's functionalities described above in this paper. Afterwards, the authoritative person requests to the citizen the National ID card in order to identify her. Such a citizen shows her National ID card, which is a new version based on our ID-NSCard. Thus, a direct and remote identification process supported by our architecture is carried out between her card and the central authoritative services (Authentication and Authorization server). Once the citizen is remotely authenticated, the

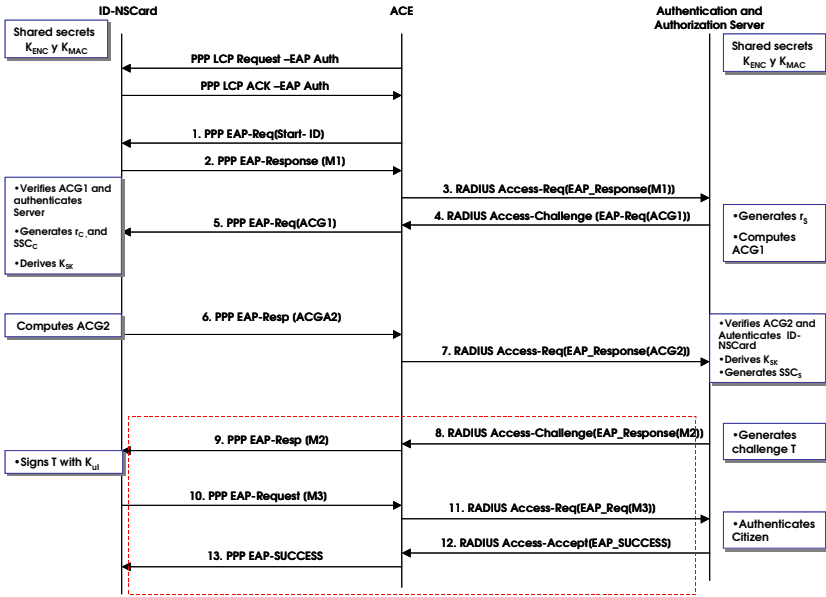


Fig. 4. Authentication flow with ID-NSCard as Spanish electronic ID-card

portable equipment could receive additional authorization information. This circumstance is out of the scope of this work. In Figure 4, the authentication flow derived from the application of our ID-NSCard to the Spanish National identification scheme is represented.

In the following paragraphs, consider the nomenclature used in the authentication protocol architecture in Figure 3.

Assume that the ACE has been correctly authenticated by the RADIUS infrastructure and that ACE and authentication server share a static key with the goal to protect their communications. Firstly, the EAP layer is activated both in the ID-NSCard and ACE by means of PPP/LCP configuration messages. Afterwards, the Spanish citizen authentication based on the ID-NSCard should take place as follows (for simplicity's sake, phase 1 is skipped):

Phase 2:

1. The ACE sends a PPP-EAP Start ID message to the ID-NSCard, in order to initiate an identification procedure based on EAP-ID.
2. The ID-NSCard returns the PPP-EAP_Response [M1] packet to the ACE, such that

$$M1 := SN_C || RND_C$$

3. The ACE encapsulates this message into a RADIUS Access-Request packet and afterwards sends it to the Authentication Server, in the back-end network.

4. Upon received M1, the Authentication server generates a random number r_s and she initiates the device authentication process by responding to the ACE with a RADIUS Access-Challenge [EAP_Response[ACG1]], such that

$$\begin{aligned}
 r_S &:= 32\text{-bytes random number} \\
 S &:= RND_S \parallel SN_S \parallel RND_C \parallel SN_C \parallel r_S \\
 ACG1 &:= E[K_{ENC}](S) \parallel MAC[K_{MAC}](E[K_{ENC}](S))
 \end{aligned}$$

5. The ACE processes the RADIUS headers and sends the received EAP packet to the ID-NSCard, encapsulated into a PPP frame.

6. After a successful verification of the authentication cryptogram ACG1 (server is authenticated), ID-NSCard generates a random number r_c and SSC_c , and derives session key K_{SK} . Afterwards, she returns the PPP-EAP _Response [ACG2] packet towards the server, such that

$$\begin{aligned}
 r_c &:= 32\text{-bytes random number} \\
 SSC_c &:= \text{send sequence counter} \\
 R &:= RND_C \parallel SN_C \parallel RND_S \parallel SN_S \parallel r_c \\
 ACG2 &:= E[K_{ENC}](R) \parallel MAC[K_{MAC}](E[K_{ENC}](R)) \\
 K_{SK} &:= r_S \oplus r_c
 \end{aligned}$$

7. The ACE builds the corresponding RADIUS-Access Request packet and sends it to the authentication server.

After a successful verification of authentication cryptogram ACG2 (ID-NSCard is authenticated), the authentication server generates SSC_s and derives the session key K_{SK} , such that

$$K_{SK} := r_S \oplus r_c$$

In this step, both devices are mutually authenticated (without cardholder participation) and they posse the session key K_{SK} , which allows them to encrypt the following communication in an end-to-end secure tunnel.

Phase 3:

8. The Authentication Server continues with this phase by sending RADIUS Access-Challenge[EAP_Response[M2]], where T is the actual protected value for challenging to the citizen. In our proposal T is a 32-bytes challenge.

$$\begin{aligned}
 T &:= SN_C \parallel RND_C \parallel RND_S \\
 M2 &:= E[K_{SK}](T)
 \end{aligned}$$

In order to provide end-to-end EAP per-packet integrity protection, note that M2 should also include the encryption of the 4-octet EAP headers (i.e. Code, Identifier and Length) and not only the encryption of the challenge T. All this information is carried in the Data field of the corresponding EAP packet. Consequently, ID-NSCard could check if the EAP headers re-transmitted by ACE correspond to the EAP headers sent by the remote authentication server.

9. The ACE processes the RADIUS headers and transmits the received EAP packet to the ID-NSCard, which is able to decrypt the message and to obtain the challenge T.

10. Once the ID-NSCard checks the freshness of T, the cardholder's password (and optionally the biometric token) is required with RSA digital signing purposes. The ID-NSCard responds to the previous challenge in a PPP-EAP Request [M3], such that

$$M3 := E_{[K_{SK}]}(\text{Sign}[K_{ul}](T) || \text{Certificate}_i)$$

As in step 8, encryption of the 4-octet EAP headers should be also included in the EAP Data field.

11. The ACE builds the RADIUS Access Request message and forwards it towards the Authentication server.

Authentication server validates the citizen's certificate and verifies her RSA signature. In this step, such a citizen is identified.

12. In case of a successful authentication, a information message is sent to the ACE and afterwards (step 13) to the smart card.

Further authorization decisions and potential services (e.g. re-authentication procedures) are out of scope of this work. With this case of study, we have shown how our authentication architecture with ID-NSCards is applied in standardized identification systems. Obviously, this architecture is easily applicable to similar national or international identification schemes, as well as, to many organizational identification systems.

7 Conclusions

Many countries are starting to issue national identity cards or electronic passports that include a chip card. This is an effective electronic way to hold a trusted identity credential by their citizens. Our work has proposed a new approach based on network smart cards with specific identification purposes, ID-NSCards. This device participates in an authentication architecture, which allows us to transport securely authentication messages between such a device and the remote authoritative server. This solution provides flexibility and robustness versus the common scheme, since the smart card behaves as an autonomous authentication supplicant, independently on the access terminal and on the characteristics of the scenario. Additionally, this solution transparently reuses the envisaged standardized authentication mechanisms for European electronic ID-Cards. Some notes about our testbed are provided and as example of application, our architecture is applied to a version of the Spanish electronic ID-Card based on our ID-NSCard. As result, multiple wired/wireless practical scenarios of utilization (both organizational and governmental) are foreseen for next future work.

References

1. EU Directive 1999/93/EC of the European Parliament and the Council of 13 December 1999 on a Community framework for Electronic Signatures (December 1999)
2. ETSI TS 101 862 v. 1.3.2: Qualified Certificate Profile (June 2004)
3. Santesson, S., Nystrom, M., Polk, T.: Internet X.509 Public Key Infrastructure: Qualified Certificates Profile, IETF RFC 3739 (March 2004)
4. CEN/CWA 14890-1, Application Interface for smart cards used as Secure Signature Creation Devices - Part 1 - Basic requirements (2004)
5. CEN/CWA 14890-2:2004; Application Interface for smart cards used as Secure Signature Creation Devices - Part 2 - Optional Features (2004)

6. CEN/CWA 15264:2005, Architecture for a European interoperable eID system within a smart card; User Requirements; Best Practice Manual for Card Scheme Operators Part 1 to 3 (2005)
7. ISO/IEC 7816-4: Identification cards - Integrated circuit(s) cards with contacts, Part 4: Inter-industry commands for interchange (2005)
8. Rees, J., Honeyman, P.: Webcard: a Java Card web server. In: Proc. of 4th IFIP Smart Card Research and Advanced Application Conference, CARDIS 2000, Bristol, U.K (2000)
9. 3GPP TS 31.111 V7.5.0, Specification of the SIM Application Toolkit (SAT) for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (September 2006)
10. Guthery, S., Kehr, R., Posegga, J.: How to Turn a GSM SIM into a Web Server. Projecting Mobile Trust onto World Wide Web. In: Proc. of 4th IFIP Smart Card Research and Advanced Application Conference, CARDIS 2000, Bristol, United Kingdom (2000)
11. Urien, P.: Internet card, a smart card as a true Internet node. *Computer Communications* 23(17), 1655–1666 (2000)
12. Postel, J.: Transmission Control Protocol, IETF RFC 079 (September 1981)
13. Itoi, N., Fukuzawa, T., Honeyman, P.: Secure Internet Smartcards. In: Attali, I., Jensen, T. (eds.) *JavaCard 2000*. LNCS, vol. 2041. Springer, Heidelberg (2001)
14. Donsez, D., Jean, S., And Lecomte, S.: Turning Multi-Applications Smart Card Services Available from Anywhere at Anytime: a SOAP/MOM approach in the context of Java Cards. In: Proc. of Smart Card Programming and Security Conference. e-Smart 2001, Cannes, France (2001)
15. Chan, A.T., Tse, F., Cao, J., Leong, H.V.: Distributed Object Programming Environment for Smart Card Application Development. In: Proc. of the Third international Symposium on Distributed Objects and Applications, September 17 - 20, 2001. IEEE Computer Society, Los Alamitos (2001)
16. Chan, A., Tse, F., Cao, J., Leong, H.V.: Enabling Distributed Corba Access to Smart Card Applications. *IEEE Internet Computing* 6(3), 27–36 (2002)
17. Chan, A.T.S., Cao, J., Chan, H., Young, G.H.: A web-enabled framework for smart card applications in health services. *Communications of the ACM* 4(9), 76–82 (2001)
18. Chan, A.T.S.: Mobile cookies management on a smart card. *Communications of the ACM* 48(11), 38–43 (2005)
19. IST Project RESET, Roadmap for European Research on Smartcard related Technologies, IST-2001-39046: Final Roadmap v.5 (May 2003)
20. Montgomery, M., Ali, A., Lu, H.K.: Secure Network Card. Implementation of a Standard Network Stack in a Smart Card. In: Proc. of 4th IFIP Smart Card Research and Advanced Application Conference, CARDIS 2004, Toulouse, France, August 23-26, 2004. Kluwer Academic Publishers, Dordrecht (2004)
21. Lu, H.K.: New Advances in Smart Card Communications, International Conference on Computing, Communications And Control technologies (CCCT), Austin, TX, USA, August 14-17 (2004)
22. Lu, H.K., Ali, A.: Prevent On-line Identity Theft - Using Network Smart Cards for Secure On-line Transactions. In: Zhang, K., Zheng, Y. (eds.) *ISC 2004*. LNCS, vol. 3225. Springer, Heidelberg (2004)
23. Ali, A., Lu, K., Montgomery, M.: Network Smart Card: A New Paradigm of Secure Online Transactions. In: Proc. of Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), Chiba, Japan, May 30 - June 1 (2005)

24. Lu, H.K.: Multi-stage Packet Filtering in Network Smart Cards. In: Domingo-Ferrer, J., Posegga, J., Schreckling, D. (eds.) *CARDIS 2006*. LNCS, vol. 3928, pp. 192–205. Springer, Heidelberg (2006)
25. Torres, J., Izquierdo, A., Sierra, J.M.: Advances in network smart cards authentication. *Computer Networks* 51(9), 2249–2261 (2007)
26. Lu, H.K.: Network smart card review and analysis. *Computer Networks* 51(9), 2234–2248 (2007)
27. Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., Levkowetz, H.: Extensible Authentication Protocol (EAP), IETF RFC 3748, Standards Track (June 2004)
28. Aboba, B., Calhoun, P.: RADIUS (Remote Authentication Dia. In: User Service) Support For Extensible Authentication Protocol (EAP), IETF RFC 3579 (September 2003)
29. Eronen, P., Hiller, T., Zorn, G.: Diameter Extensible Authentication Protocol (EAP) Application, IETF RFC 4072 (August 2005)
30. Simpson, W.: The Point-to-Point Protocol (PPP), IETF RFC 1661, Standard Track (July 1994)
31. FreeRADIUS, GNU General Public License, <http://www.freeradius.org>
32. JRadius-Client, SourceForge Project, <http://jradius-client.sourceforge.net>
33. ETSI TS 101 456 v.1.2.1, Policy Requirements for certification authorities issuing qualified certificates (April 2002)
34. ETSI TS 102 042 v.1.1.1, Policy Requirements for certification authorities issuing public key certificates (April 2002)

SmartPRO: A Smart Card Based Digital Content Protection for Professional Workflow

Alain Durand, Marc Éluard, Sylvain Lelievre, and Christophe Vincent

Thomson R&D France

Technology Group, Corporate Research, Security Laboratory
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
{alain.durand,marc.eluard,sylvain.lelievre,
christophe.vincent}@thomson.net

Abstract. This paper introduces SmartPRO, a smart card based technology aiming at protecting content in professional workflows. It gives an overview on how SmartPRO works. It also explains the design constraints that led to the use of smart cards and some of the extra difficulties implied by this choice in order to get to an implementation that may be industrially deployed.

1 Introduction

Digital Rights Management (DRM) has been for a decade a widely studied subject. Traditional goal for a DRM is to prevent an end-user to make an unauthorized use of a piece of content (usually music or video). Piracy of digital content has actually been a growing issue since the entrance in the digital era and the widespread of high-speed communications. Black Market for DVDs are now important in most countries (see e.g. [1]). The MPAA (Motion Picture Association of America), the association of the seven major Hollywood studios, estimated to \$6.1 billion the cost of video piracy in 2005 [2].

Generally, movie distribution obeys to different diffusion windows: film is first distributed in theaters, then in hotels or planes and DVD release occurs right after. It is then distributed to television first on Pay-Per-View or Video-on-Demand systems, then on Pay-TV and eventually on Free-To-Air channels. Table 1 shows average breakdown of movie revenues along the different diffusion windows.

Different solutions protect the release windows shown on Table 1. For instance, AAC3 [3] or CSS [4] protect home video / rental window while Conditional Access systems protect Pay TV or Cable TV window. DCI (Digital Cinema Initiative) specification [5] includes a protection scheme for digital theaters. Some systems (e.g., broadcast flag [6] or CPCM (Copy Protection and Content Management [7])) protect content distributed in the syndication window.

One could thus think that content protection technologies coverage is sufficient. This is however not the case. Roughly 10% of revenues loss is due to piracy operated before the content release in theaters [8]. This happened for instance to the recent Ridley Scott movie, American Gangster [9].

Table 1. Movie Revenues (Courtesy of Technicolor)

	Theatrical Release	Airline/Hotel	Home Video/Rental	PPV/VOD inDemand, DirectTV	Pay TV HBO, Showtime	Networks Cable TV	Syndication
Time Frame (in month)	0	2-4	4-6 (ongoing)	6-9 (only for 30-45 days)	12-15	24-30	36-42
Typical/Approximate Revenues (\$10M + box-office movie)	25%	1%	56%	2.5%	10%	3%	2.5%

The traditional approach to overcome this threat is based on network security technologies and on physical access control to facilities. While these techniques may be efficient against external attackers, this is not the case for insiders. This happened for instance for the third episode of Star Wars [10] that has been actually stolen from post-production facilities. It then passed through several go-betweens before being eventually made available on the Internet.

SmartPRO technology is a content protection system for professional workflows. Its first real deployment aimed to prevent leakage from video production and post-production facilities. SmartPRO is a smart card based technology that is for example deployed in Nexguard CP [11] product line. Smart card offers a secure place to store system keys and guarantees the integrity of the software using these keys. Any other transportable security token could be also used.

The main underlying idea behind this technology is to upgrade technologies or techniques that have been successful to protect the content in the consumer space to the professional realm. We propose to base the system on smart cards as Conditional Access systems do and we adapted the notion of consumer domain (see for instance [7]) to enable collaborative work.

The rest of the paper is organized as follows. In the next section, we give an overview of SmartPRO technology. Next we explain how smart cards are used in SmartPRO and give some design rationales. Finally, we present examples of difficulties when designing the system using secure processor.

2 General Presentation

SmartPRO introduces the notion of Virtual Domain (VD). A VD is a set of devices that can share private contents. It can represent a company or part of a company like a post production facility. A VD is not bound to a person or a physical location. It can be used for any content format and using any network technology or physical interface.

Inside a VD, content is scrambled, i.e. encrypted, with a cryptographic key¹. This key is protected so that only devices belonging to the VD can access to

¹ In the video content industry, the term scrambling is typically used rather than encryption for content protection. This term make references to the first mechanisms for Pay-TV where some parts of the content where re-ordered to achieve protection.

it, and thus to the content. SmartPRO mainly brings a key management system implementing this notion of Virtual Domain.

2.1 Actors in Virtual Domain

Figure 1 illustrates the basic elements of a VD. Acquisition devices are the entry points of the VD. From that point on, the content is digital and protected. It can be accessed, if allowed, by any renderer devices, or processing devices of the VD.

Renderer devices are the final points of the VD. After a renderer device, content does not benefit anymore from SmartPRO protection. Special care will be needed within the renderer devices to avoid theft of content once it is no longer SmartPRO protected. Content may be in the clear or protected by another copy protection scheme (for example with watermark).

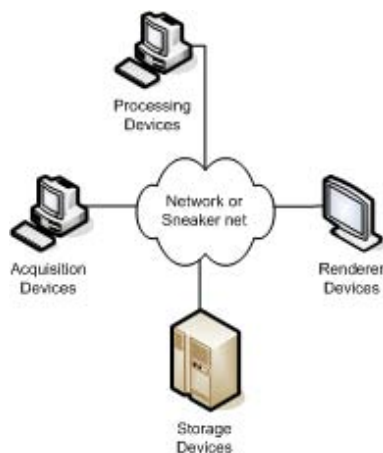


Fig. 1. Devices in a Virtual Domain

Processing devices modify SmartPRO content. Prior to applying the expected process, device unprotects the content or part of it. Once processed, the device reprotects the content. All these operations are performed in a secure environment (i.e. content remains in the same VD). SmartPRO processing devices cannot create SmartPRO protected content from clear content.

Storage devices do not act on the content at all. They are simple bit buckets.

2.2 Content Protection

SmartPRO protected content is always scrambled. Scrambling mechanism is based on robust cryptographic algorithm (AES-CTR [12]). The scrambling keys are called Control Words (CW). Usage and access rules of the SmartPRO protected content are called Usage Rights. Control Words and usage rights are

embedded in licenses called Local Enforcement Copy Management (LECM). A given license is valid only in one Virtual Domain for only one content. License is created in the acquisition device at the same time as the content is acquired. Licenses are analyzed and enforced by devices before they handle the content. A processing device can modify the license if granted in the usage rights.

License is partially encrypted with a secret key called Domain Key (to ensure confidentiality of CW) and signed (to ensure integrity of LECM). This key is unique to a VD and is randomly generated at VD creation. All devices belonging to a given VD share its Domain Key. A device belongs to only one VD.

2.3 Multiple Virtual Domains

A simple VD may not be sufficient for many cases. For instance, two firms *A* and *B* may use SmartPRO. Each manages its own VD. In some cases, some protected data may be transferred from domain *A* to domain *B* but data must remain protected during this transfer. Thus, SmartPRO introduces the notion of Multiple VDs. Content is able to flow through controlled devices, called Bridge devices, from one VD to another VD. The owner of the source VD manages the usage rights of the delivered content to the destination VD. Figure 2 illustrates this architecture.

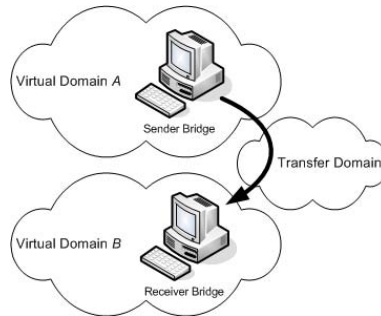


Fig. 2. Example of multiple Virtual Domains

The bridging operation between two VDs is performed with two entities: The Sender Bridge prepares and sends a content from source domain into a transfer domain. The Receiver Bridge receives and transfers a content from the transfer domain to the destination domain. This transfer domain is temporary and completely transparent for the user. Content remains unchanged (i.e. scrambled) during the operation and only the licenses are processed. Usage Rights associated to the content may be modified by the Sender Bridge.

Sender Bridge is able to send content to multiple VDs. Receiver Bridge is only able to send back the content to the source domain.

3 Device: A Collaboration between Host and Token

3.1 Overview

Since no software solution can be considered sufficiently secure, the management of sensitive operations and secret data should be done by secure hardware. We choose Smart cards (called tokens) as secure hardware to store secret data and run applications in a tamper resistant environment. Nevertheless, their processors do not permit to encrypt or decrypt large amount of data in a reasonable time.

Thus, it is preferable to use a host with a powerful processor associated to a token. The content is scrambled or descrambled by the host while the license is managed by the token.

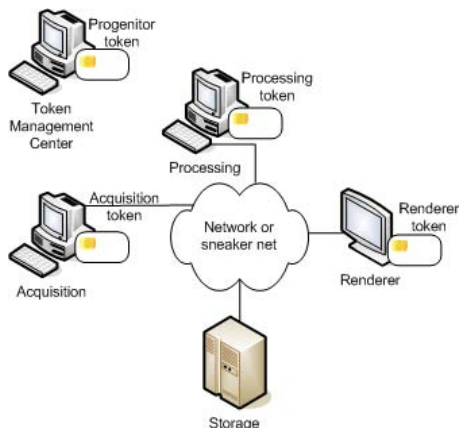


Fig. 3. Architecture of Virtual Domain with tokens

To keep the domain key as secure as possible, it is managed by the token and never leaves it. However the control words that protect content need to be used by tokens and also by hosts. If an attack succeeds on a host, only control word may leak and so only the corresponding content may be broken. The other contents of the domain remain secure. If an attacker targets all contents of a domain, he must extract the domain key from the token.

The host is not bound to any domain until a token is inserted. The host is then temporary bound to the token's domain. So, only the tokens belong to a Virtual Domain.

3.2 Token Management Center

Token Management Center (TMC) has a major role in key management. It performs the enrollment or activation of tokens in a Virtual Domain. A special token, the progenitor token, is associated to each Virtual Domain. There is only one progenitor in a Virtual Domain. The activation of a new token to a VD requires the presence of its progenitor token in the TMC.

TMC runs on standard computers. It supports simultaneously at least two tokens. The TMC does not store any secret. The progenitor token handles all the secrets including the domain key.

All progenitor token are delivered inactive to the user and activated using TMC. During this activation, the progenitor generates a domain key and securely stores it. During the activation of a token, the progenitor securely transfers the domain key to the token.

SmartPRO supports revocation of domains, tokens and hosts (see section 4.1).

The progenitor token creates and maintains a database to store the serial numbers of the tokens that has been activated or revoked in the domain.

The TMC does not need to be online with any device of the VD, or any back-office. Nevertheless, online connection with devices may allow remote management of tokens.

3.3 Hosts and Tokens Interactions

Once the token is linked to a domain, it can deal with SmartPRO content. In an acquisition device, the host receives clear content to be scrambled, it requests the token to build a license. The token picks up a random control word. It inserts the CW in the license and encrypts the license with the domain key. Then, token sends both CW and license to the host. The host scrambles the content with the CW.

To descramble a protected content, a renderer device needs the license corresponding to the content. The host sends the license to the token. If the token and the content are in the same domain, the license can be decrypted using the domain key. Then, the decrypted CW can be sent to the host to descramble the content.

The domain key never leaves the token, only CW is provided to the host. The token shall first ensure that the host is trustful, and compliant. Non-Compliant hosts could divulgate the CW. A compliant host, even purely software, protects the CW and the content upon its descrambling. Hence, some secure coding techniques (e.g.: code obfuscation, anti-debugger) are used to make difficult to modify host behavior. Furthermore, before sending the CW to the host, the token authenticates the host. To that end, the host needs to have a public/private key pair. Finally, the CW needs to be sent encrypted since communications between the token and the host are easy to eavesdrop. For these reasons, a Secure Authenticated Channel (SAC) is setup prior to any communication.

3.4 Secure Authenticated Channel

The SAC is used by progenitor token during token activation or token deactivation and for CW transmission to the host (see section 3.3).

The protocol is based on Diffie-Hellman [13]. Each entity is given random private key (K_{priv}) and a certificate that embeds an identity (e.g., the certificate serial number) and the public key ($K_{pub} = g^{K_{priv}} \bmod p$ where g and p are Diffie-Hellmann parameters shared by all entities). For the sake of simplicity, the notation $\bmod p$ will be omitted in the rest of the document but it shall be understood that all exponentiations of g are performed modulo p .

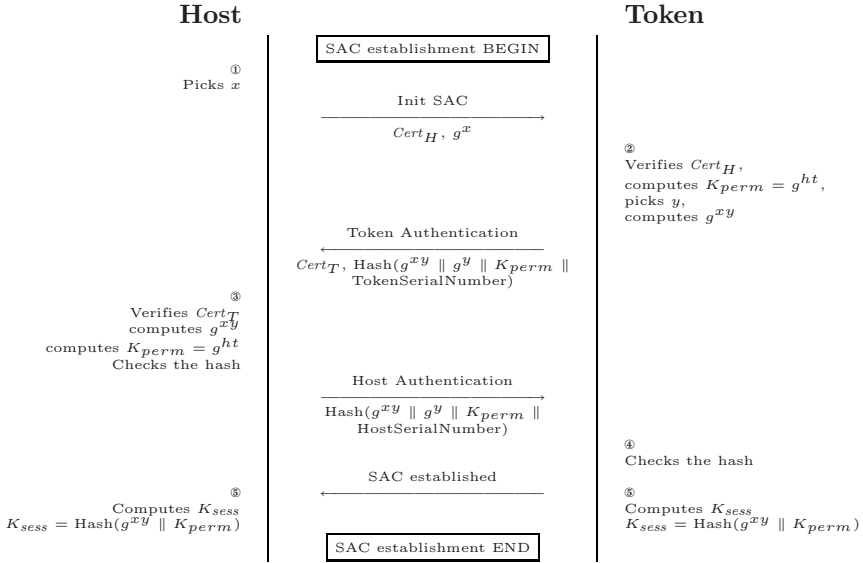


Fig. 4. Secure Authenticated Channel Protocol

1. The host picks a random x , computes the associated public value g^x and sends the result to the token together with its certificate $Cert_H$.
2. The token extracts the public key g^h from the host certificate. It verifies that the certificate is valid. It then computes secret key $K_{perm} = g^{ht}$ where t is the token certificate secret key. The token also picks a random y , computes the associated public value g^y . It computes as well the hash value of the concatenation of g^y , g^{xy} , K_{perm} and its serial number. It sends the result of both computations together with its certificate to the host.
3. The host extracts the public key g^t and verifies that the certificate is valid. It then computes secret key $K_{perm} = g^{ht}$ and g^{xy} . It also checks whether the received hash value is correct. It computes then the hash value of the concatenation of g^x , g^{xy} , K_{perm} and its serial number and sends the result to the token.
4. The token verifies the correctness of received hash value.
5. Both token and host compute the session key K_{sess} as the hash value of g^{xy} and K_{perm} . K_{sess} will be used to secure further communication between the host and the token.

4 Using Secure Processor

4.1 Revocation Mechanism

The security of SmartPRO is based on a removable secure processor (a token). It guarantees that all the secret data are securely stored and processed.

Nevertheless, we know that no security system is 100% secure. Hackers use more and more sophisticated tools that will eventually defeat any security mechanism. Thus, it is important to have a revocation mechanism that will prevent a compromised element from working. In case of major hack, the replacement of all tokens should be planned.

The revocation mechanism defined in the SmartPRO specification is based on two revocation lists:

- The Internal Revocation List (IRL) contains the elements revoked in a given domain and is managed (created and updated) by the domain manager (progenitor token). The IRL only addresses token elements and contains the Certificate Serial Number (SN) of the revoked tokens.
- The External Revocation List (ERL) contains the elements revoked in all SmartPRO system. The ERL addresses token, host and VD elements. The ERL contains serial number of host and token, and Virtual Domain identifier (VDID) for Virtual Domain.

Revocation List Usage. During the first messages of the SAC establishment, the host and the token exchange their certificate. At this moment, each entity checks that the serial number of their peers certificate is not present in the revocation list. If so, the SAC establishment continues as specified in Section 3.4

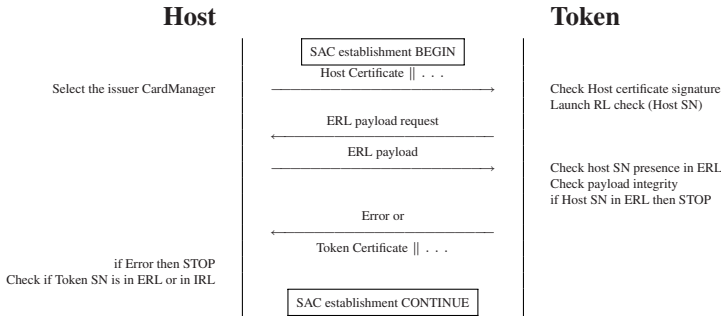


Fig. 5. Revocation List Usage Protocol

The difficulties to implement our revocation mechanism were:

- The token has a given limited amount of memory. It cannot store an ever increasing list.
- The token and the host must always hold the same version of the lists.

Revocation List Format. Each list has a header and a payload. The header contains an ever increasing index of the list, and for each element type (host, token or VD), the number of revoked elements and a digest (SHA-1) of the list of revoked elements. The header is signed by a root revocation key for the ERL and the progenitor revocation key for the IRL. The payload gives for each element type, the SN (or VDID for virtual domain) of the revoked elements.

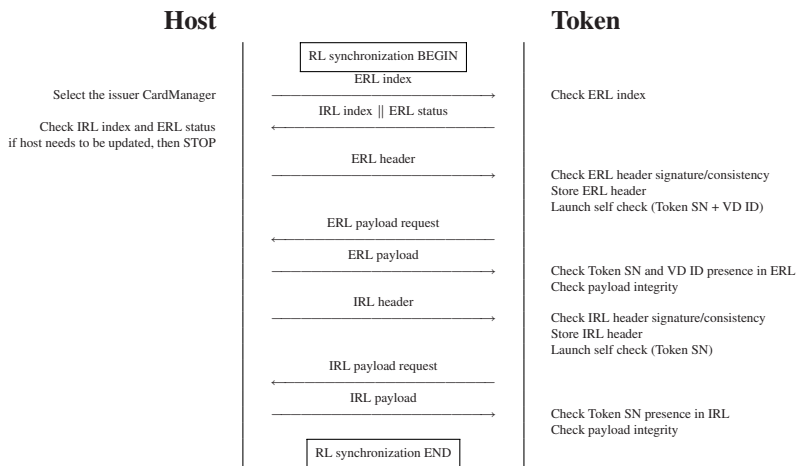


Fig. 6. Revocation List Synchronization Protocol

The lists indexes have been integrated in the messages exchanged between the host and the token during the SAC establishment. Thus, the SAC will not be established if the host and the token are not synchronized on the same lists.

Revocation List Storage. The token only stores the list headers. When it receives its host certificates (during the SAC establishment), it requests the list to the host prior to checking if the certificate is in the list. It checks the validity of the received list using the digest value contained in the header.

Revocation List Synchronization. The host sends to the token its external RL index and the token responds with its internal RL index and an External Status indicating if an update is needed. If the token needs an update, the host sends the new RL header. If the host needs an update, it must retrieve the new RL before any further collaboration with the token.

4.2 Bridging Implementation

Another key issue in our implementation was the bridging mechanism. It transfers content from a source domain to one or several other destination domains. One solution would be to send the license of the domain source to the relay token which would decrypt it and re-encrypt it for each destination domain. This means that the relay token should contain secret keys of all the potential destination domains!

Our solution uses two kinds of host/token:

- The Master Bridge (MB, host and token): It only knows source Domain Key. It converts the license for the source domain into a license for a transfer domain. It then generates descrambling information specific for each destination domains (CDI for Content Descrambling Information). The Master

Bridge token is initialized in the source domain. It holds a certificate containing its serial number (MB SN).

- The Simple Bridge (SB, host and token): It processes the license from the transfer domain and generates a license for a destination domain. A Simple Bridge token is initialized twice. First it is activated in the destination domain. Then it must be registered in the source domain: It receives the information needed to process the CDI generated by the Master Bridge token. These information include an initialization index. This index is ever increasing in the source domain and is used by the Master Bridge token to generate the CDI. The Simple Bridge holds a certificate containing its serial number (SB SN).

The source domain progenitor generates and manages the following elements:

- A master key for Master bridge (MK_{MB}) used to calculate derived key for Master bridge (DK_{MB}): $DK_{MB} = E\{MK_{MB}\}(MB\ SN)$.
- A master key for Simple Bridge (MK_{SB}) used to calculate derived Key for Simple Bridge (DK_{SB}): $DK_{SB} = E\{MK_{SB}\}(SB\ SN \parallel SB\ index)$.
- The Authorization Mask is a bit mask where the position of each bit corresponds to a SB index. If the bit is "1", the corresponding SB is registered and not revoked in the IRL of the source domain. The Authorization Mask is generated and updated by the progenitor and signed with the progenitor revocation private key. The Authorization Mask is transmitted to all Master Bridge hosts. The Authorization Mask will be used by the Master Bridge to know if it can generate CDI for a given Simple Bridge. The use of Authorization Mask simplifies the operation in the Master bridge token: all checks relative to Simple Bridge registration or revocation are performed by the progenitor.

The Master Bridge token holds the following information received from the progenitor during its activation:

- MK_{SB} ,
- DK_{MB} calculated by the progenitor.

The Master Bridge token also stores the latest version of the Authorization Mask. The update of the Authorization Mask is performed by the Master Bridge host before any bridging operation in order to take into account new registered or revoked Simple Bridge tokens.

The Simple Bridge token holds the following information received from the progenitor during its registration:

- MK_{MB} ,
- DK_{SB} calculated by the progenitor,
- an initialization index.

On the Master Bridge side During a bridging operation, the Master Bridge host sends the license of the content to its token. The license is converted into a

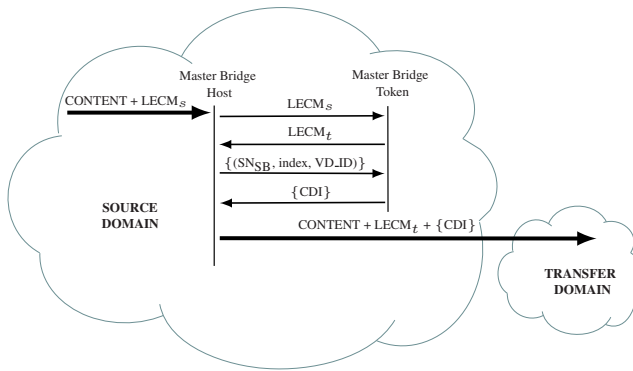


Fig. 7. Master Bridge

license for a transfer domain, encrypted by a transfer license key (TK_{LECM}). Then, for each destination domain, information on the Simple Bridge is sent to the Master Bridge token (SB SN, SB index and VD_ID). The Master Bridge token creates CDI. A CDI contains TK_{LECM} encrypted with a bridge key (BK). The Bridge Key is calculated from DK_{MB} and DK_{SB} : $BK = DK_{MB} \parallel DK_{SB}$. The Master Bridge token calculates DK_{SB} with MK_{SB} and the Simple Bridge token information. The CDI also contains the serial number of the targeted Simple Bridge token. The scrambled content, the transfer license and the list of CDIs are transmitted to each SB device.

On the Simple Bridge Side When receiving these data, the Simple Bridge host sends the license and its CDI to the Simple Bridge token. The token calculates DK_{MB} with MK_{MB} and the Master Bridge token information. It can then calculate the bridge key and retrieve TK_{LECM} . It then converts the transfer license into a license for its domain.

The content is not processed and remains scrambled during the bridging operation.

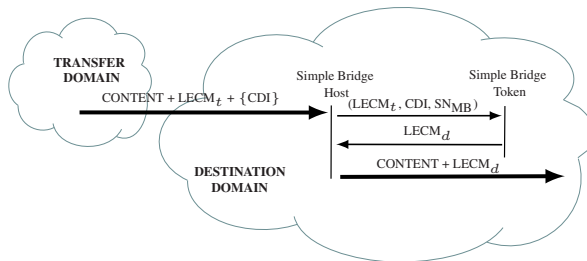


Fig. 8. Simple Bridge

5 Conclusion

SmartPRO is a content protection scheme preventing content leakage in professional workflows. A main design criteria was to achieve high security and renewability. Hence, the choice of a smart card based implementation was straightforward.

This choice however led to a greater system complexity due to lack of computational power and bandwidth of smart cards. We had to design original mechanisms to deal with flexible cards and hosts revocation.

SmartPRO only cares about basic layers of content protection. Other techniques may be plugged in the upper layers. For example, Nexguard Content Protection [11] uses watermarking technology allowing to trace back the origin of the content leakage. Adding a rights expression language or an access control technology would allow to further control the distribution of the protected content, e.g., by setting a content license expiration time or a user-based access granularity. Whatever the technology plugged above SmartPRO is, the system designer will face the same constraints to use adequately the protection offered by the smart card without impeding the whole system.

References

1. Fake DVD seizures up 41% on 2004(last visited May 2008), <http://news.bbc.co.uk/1/hi/entertainment/film/4099696.stm>
2. MPA 2005 US piracy fact sheet (last visited February 2008), <http://www.mpaa.org/USPiracyFactSheet.pdf>
3. Advanced Access Content System for Pre-recorded Book (AACSB), v0.91, February 17, 2006 (last visited February 2008), <http://www.aacsla.com/specifications/>
4. CSS Description (last visited February 2008), http://en.wikipedia.org/wiki/Content_Scramble_System
5. DCI Specifications (last visited February 2008), <http://www.dcmovies.com/specification/index.tt2>
6. Broadcast Flag Description (last visited February 2008), http://en.wikipedia.org/wiki/Broadcast_flag
7. CPCM Description (last visited February 2008), <http://www.dvb.org/technology/dvb-cpcm/>
8. Byers, S., Cranor, L., Cronin, E., Kormann, D., McDaniel, P.: Analysis of security vulnerabilities in the movie production and distribution process. In: ACM workshop on Digital rights management, October 27 (2003)
9. <http://o.seattletimes.nwsource.com/html/movies/2004016889-gangster16.html> (last visited February 2008)
10. <http://news.bbc.co.uk/1/hi/entertainment/4650956.stm> (last visited February 2008)
11. NexGuard Content Protection (last visited February 2008), <http://www.thomson.net/GlobalEnglish/Products/content-tracking-and-security/nexguard/nexguard-content-protection/Pages/default.aspx>
12. Daemen, J., Rijmen, V.: The design of Rijndael: AES – the Advanced Encryption Standard. Springer, Heidelberg (2002)
13. Diffie, W., Hellman, M.E.: New Directions in Cryptography. IEEE Transactions on Information Theory IT-22(6), 644–654 (1976)

A Practical Attack on the MIFARE Classic

Gerhard de Koning Gans, Jaap-Henk Hoepman,
and Flavio D. Garcia

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
gkoningg@sci.ru.nl,
jhh@cs.ru.nl,
flaviog@cs.ru.nl

Abstract. The MIFARE Classic is the most widely used contactless smart card in the market. Its design and implementation details are kept secret by its manufacturer. This paper studies the architecture of the card and the communication protocol between card and reader. Then it gives a practical, low-cost, attack that recovers secret information from the memory of the card. Due to a weakness in the pseudo-random generator, we are able to recover the keystream generated by the CRYPTO1 stream cipher. We exploit the malleability of the stream cipher to read *all* memory blocks of the first sector of the card. Moreover, we are able to read *any* sector of the memory of the card, provided that we know *one* memory block within this sector. Finally, and perhaps more damaging, the same holds for *modifying* memory blocks.

1 Introduction

RFID and contactless smart cards have become pervasive technologies nowadays. Over the last few years, more and more systems adopted this technology as replacement for barcodes, magnetic stripe cards and paper tickets for a variety of applications. Contact-less cards consist of a small piece of memory that can be accessed wirelessly, but unlike RFID tags, they also have some computing capabilities. Most of these cards implement some sort of simple symmetric-key cryptography, which makes them suitable for applications that require access control.

A number of high profile applications make use of contactless smart cards for access control. For example, they are used for payment in several public transport systems like the Octopus card¹ in Hong Kong, the Oyster card² in London, and the OV-Chipkaart³ in The Netherlands, among others. Many countries have already incorporated a contactless card in their electronic passports³ and several car manufacturers have it embedded in their car keys as an anti-theft method.

¹ <http://www.octopuscards.com/>

² <http://oyster.tfl.gov.uk>

³ <http://www.ov-chipkaart.nl/>

Many office buildings and even secured facilities like airports and military bases, use contactless smart cards for access control.

On the one hand, the wireless interface has practical advantages: without mechanical components between readers and cards, the system has lower maintenance costs, is more reliable, and has shorter reading times, providing higher throughput. On the other hand, it represents a potential threat to privacy [3] and it is susceptible to relay, replay and skimming attacks that were not possible before.

There is a huge variety of cards on the market. They differ in size, casing, memory and computing power. They also differ in the security features they provide. A well known and widely used system is MIFARE. MIFARE is a product family from NXP semiconductors (formerly Philips). According to NXP there are about 200 million MIFARE cards in use around the world, covering 85% of the contactless smartcard market. The MIFARE family contains four different types of cards: Ultralight, Standard, DESFire and SmartMX. The MIFARE Classic cards come in three different memory sizes: 320B, 1KB and 4KB. The MIFARE Classic is the most widely used contactless card in the market. Throughout this paper we focus on this card. MIFARE Classic provides mutual authentication and data secrecy by means of the so called CRYPTO1 stream cipher. This cipher is a proprietary algorithm of NXP and its design is kept secret.

Nohl and Plötz [7] have recently reverse engineered the hardware of the chip and exposed several weaknesses. Among them, due to a weakness on the pseudo-random generator, is the observation that the 32-bit nonces used for authentication have only 16 bits of entropy. They also noticed that the pseudo-random generator is stateless. They claim to have knowledge of the exact encryption algorithm which would facilitate an off-line brute force attack on the 48-bit keys. Such an attack would be feasible, in a reasonable amount of time, especially if dedicated hardware is available.

Our Contribution. We used a Proxmark III⁴ to analyze MIFARE cards and mount an attack. To do so, we have implemented the ISO 14443-A functionality on the Proxmark, since only ISO 14443-B was implemented at that time. We programmed both processing and generation of reader-to-tag and tag-to-reader communication at physical and higher levels of the protocol. The source code of the firmware is available in the public domain⁵. Concurrently, and independently from Nohl and Plötz results, we also noticed a weakness in the pseudo-random generator.

Our contribution is threefold: First and foremost, using the weakness of the pseudo-random generator, and given access to a particular MIFARE card, we are able to recover the keystream generated by the CRYPTO1 stream cipher, without knowing the encryption key. Secondly, we describe in detail the communication between tag and reader. Finally, we exploit the malleability of the stream cipher to read *all* memory blocks of the first sector (sector zero) of the card (without having access to the secret key). In general, we are able to read

⁴ <http://cq.cx/proxmark3.pl>

⁵ <http://www.proxmark.org>

any sector of the memory of the card, provided that we know *one* memory block within this sector. After eavesdropping a transaction, we are always able to read the first 6 bytes of every block in that sector, and in most cases also the last 6 bytes. This leaves only 4 unrevealed bytes in those blocks.

We would like to stress that we notified NXP of our findings before publishing our results. Moreover, we gave them the opportunity to discuss with us how to publish our results without damaging their (and their customers) immediate interests. They did not take advantage of this offer.

Consequences of Our Attack. Any system using MIFARE Classic cards that relies on the secrecy or the authenticity of the information stored on sector zero is now insecure. Our attack recovers, in a few minutes, *all* secret information in that sector. It also allows us to *modify* any information stored there. This is also true for most of the data in the remaining sectors, depending on the specific scenario. Besides, our attack complements Nohl and Plötz results providing the necessary plaintext for a brute force attack on the keys. This is currently work in progress.

Outline of this Paper. Section 2 describes the architecture of the MIFARE cards and the communication protocol. Section 3 describes the hardware used to mount the attack. Section 4 discusses the protocol by a sample trace. Section 5 exposes weaknesses in the design of the cards. The attack itself is described in Section 6. Finally, Section 8 gives some concluding remarks and detailed suggestions for improvement.

2 MIFARE Classic

Contactless smartcards are used in many applications nowadays. Contactless cards are based on *radio frequency identification* technology (RFID) [1]. In 1995 NXP, Philips at that time, introduced MIFARE⁶. Some target applications of MIFARE are public transportation, access control and event ticketing. The MIFARE Classic [8] card is a member of the MIFARE product family and is compliant with ISO 14443 up to part 3. ISO 14443 part 4 defines the high-level protocol and here the implementation of NXP differs from the standard. Section 2.1 discusses the different parts of the ISO standard.

2.1 Communication Layer

The communication layer of the MIFARE Classic card is based on the ISO 14443 standard [4]. This ISO standard defines the communication for identification cards, contactless integrated circuit(s) cards and proximity cards. The standard consists of four parts.

Part 1 describes the physical characteristics and circumstances under which the card should be able to operate.

⁶ <http://www.nxp.com>

Part 2 defines the communication between the reader and the card and vice versa. The data can be encoded and modulated in two ways, type A and type B. MIFARE Classic uses type A. For more detailed information about the communication on RFID we refer to the “RFID Handbook” by Klaus Finkenzeller [1].

Part 3 describes the initialization and anticollision protocol. The *anticollision* is needed in order to select a particular card when more cards are present within the reading range of the reader. After a successful initialization and anticollision the card is in an active state and ready to receive a command.

Part 4 defines how commands are send. This is the point where MIFARE Classic differs from the ISO standard, using a proprietary and undisclosed protocol. The MIFARE Classic starts with an authentication, after that all communication is encrypted. On every eight bits a parity bit is computed to detect transmission errors. In the MIFARE Classic protocol this parity bit is also encrypted which means that integrity checks are only possible in the application layer.

2.2 Logical Structure

A MIFARE Classic card is in principle a memory card with few extra functionalities. The memory is divided into data blocks of 16 bytes. Those data blocks are grouped into sectors. The MIFARE Classic 1k card has 16 sectors of 4 data blocks each. The first 32 sectors of a MIFARE Classic 4k card consists of 4 data blocks and the remaining 8 sectors consist of 16 data blocks. Every last data block of a sector is called *sector trailer*. A schematic of the memory of a MIFARE Classic 4k card is shown in Figure 1.

Note that block 0 of sector 0 contains special data. The first 4 data bytes contain the unique identifier of the card (UID) followed by its 1-byte *bit count check* (BCC). The bit count check is calculated by successively XOR-ing all UID

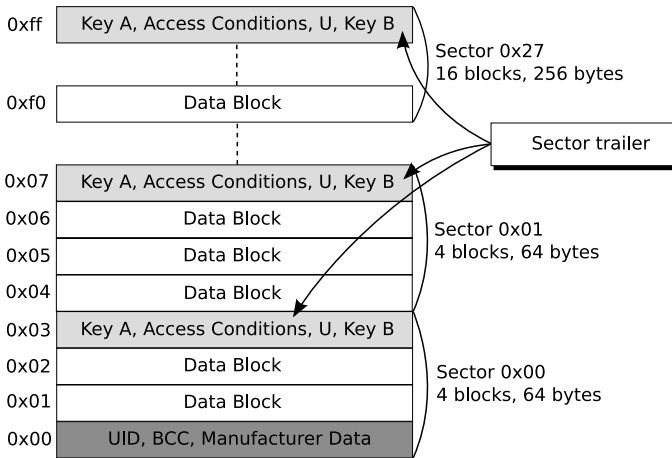


Fig. 1. MIFARE Classic 4k Memory

bytes. The remaining bytes are used to store manufacturer data. This data block is read-only. The reader needs to authenticate for a sector before any memory operations are allowed. The sector trailer contains the secret keys *A* and *B* which are used for authentication. The *access conditions* define which operations are available for this sector.

The sector trailer has special access conditions. Key *A* is never readable and key *B* can be configured as readable or not. In that case the memory is just used for data storage and key *B* cannot be used as an authentication key. Besides the access conditions (AC) and keys, there is one data byte (*U*) remaining which has no defined purpose. A schematic of the sector trailer is shown in Figure 2a. A data block is used to store arbitrary data or can be configured as a *value block*. When used as a value block a signed 4-byte value is stored twice non-inverted and once inverted. Inverted here means that every bit of the value is XOR-ed with 1. These four bytes are stored from the least significant byte on the left to the most significant byte on the right. The four remaining bytes are used to store a 1-byte block address that can be used as a pointer.

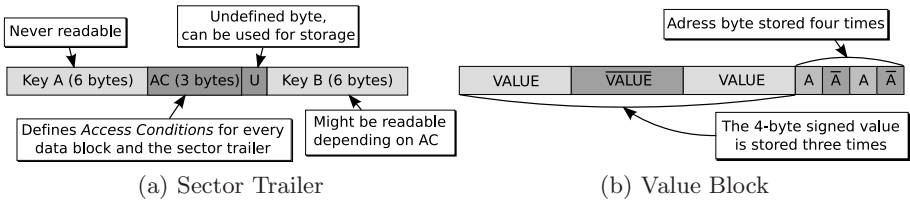


Fig. 2. Block contents

2.3 Commands

The command set of MIFARE Classic is small. Most commands are related to a data block and require the reader to be authenticated for its containing sector. The access conditions are checked every time a command is executed to determine whether it is allowed or not. A block of data might be configured to be read only. Another example of a restriction might be a value block which can only be decremented.

Read and Write. The read and write commands read or write one data block. This is either a data block or a value block. The write command can be used to format a data block as value block or just store arbitrary data.

Decrement, Increment, Restore and Transfer. These commands are only allowed on data blocks that are formatted as value blocks. The *increment* and *decrement* commands will increment or decrement a value block with a given value and place the result in a memory register. The *restore* command loads a value into the memory register without any change. Finally the memory register is transferred in the same block or transferred to another block by the *transfer* command.

2.4 Security Features

The MIFARE Classic card has some built-in security features. The communication is encrypted by the proprietary stream cipher CRYPTO1.

Keys. The 48-bit keys used for authentication are stored in the sector trailer of each sector (see section 2.2). MIFARE Classic uses symmetric keys.

Authentication Protocol. MIFARE Classic makes use of a mutual three pass authentication protocol that is based on ISO 9798-2 according to the MIFARE documentation [8]. However, it turned out that this is not completely true [2]. In this paper we only use the first initial nonce that is send by the card. The reader sends a request for sector authentication and the card will respond with a 32-bit nonce N_C . Then, the reader sends back an 8-byte answer to that nonce which also contains a reader random N_R . This answer is the first encrypted message after the start of the authentication procedure. Finally, the card sends a 4-byte response. As far as our attack is concerned this description captures all the necessary information.

3 Hardware and Software

An RFID system consists of a transponder (card) and a reader [1]. The reader contains a radio frequency module, a control unit and a coupling element to the card. The card contains a coupling element and a microchip. The control unit of a MIFARE Classic enabled reader is typically a MIFARE microchip with a closed design. This microchip communicates with the application software and executes commands from it. Note that the actual modulation of commands is done by this microchip and not by the application software. The design of the microchip of the card is closed and so is the communication protocol between card and reader.

We want to evaluate the security properties of the MIFARE system. Therefore we need hardware to eavesdrop a transaction. It should also be possible to act like a MIFARE reader to communicate with the card. The Proxmark III developed by Jonathan Westhues has these possibilities [7]. Because of its flexible design, it is possible to adjust the Digital Signal Processing to support a specific protocol. This device supports both low frequency (125 kHz - 134 kHz) and high frequency (13.56 MHz) signal processing. The signal from the antenna is routed through a Field Programmable Gate Array (FPGA). This FPGA relays the signal to the microcontroller and can be used to perform some filtering operations before relaying. The software implementation allows the Proxmark to eavesdrop communication (Figure 4) between an RFID tag

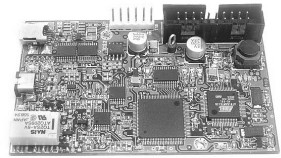


Fig. 3. The Proxmark III

⁷ Hardware design and software is publicly available at <http://cq.cx/proxmark3.pl>

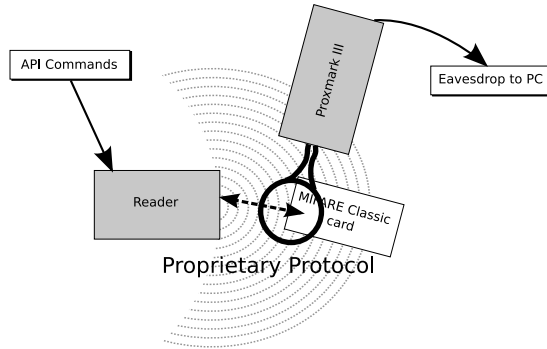


Fig. 4. Experimental Setup

and a reader, emulate a tag and a reader. In this case our tag will be the MIFARE Classic card. Despite the basic hardware support for these operations the actual processing of the digitized signal and (de)modulation needs to be programmed for each specific application. The physical layer of the MIFARE Classic card is implemented according to the ISO14443-A standard [4]. We had to implement the ISO14443-A functionality since it was not implemented yet. This means we had to program both processing and generation of reader-to-tag and tag-to-reader communication in the physical layer and higher level protocol. To meet the requirements of a replay attack we added the functions ‘hi14asnoop’ to make traces, ‘hi14areader’ to act like a reader and ‘hi14asim’ to simulate a card. We added the possibility to send custom parity bits. This was needed because parity bits are part of the encryption.

4 Communication Characteristics

To find out what the MIFARE Classic communication looks like we made traces of transactions between MIFARE readers and cards. This way, we gathered many traces which gave us some insights on the high-level protocol of MIFARE Classic. In this section we explain a trace we recorded as an example, which is shown in Figure 5. This trace contains every part of a transaction. We refer to the sequence number (SEQ) of the messages we discuss. The messages from the reader are shown as PCD (Proximity Coupling Device) messages and from the card as TAG messages. The time between messages is shown in Elementary Time Units (ETU). One ETU is a quarter of the bit period, which equals $1.18 \mu\text{s}$. The messages are represented in hexadecimal notation. If the parity bit of a byte is incorrect⁸, this is shown by an exclamation mark. We will discuss only the most significant messages.

Anticollision. The reader starts the SELECT procedure. The reader sends 93 20 (#3), on which the card will respond with its unique identifier (#4). The

⁸ Encrypted parity bits show up as parity error in the message.

ETU SEQ	sender	bytes	
0 : 01 :	PCD	26	} Anticollision
64 : 02 :	TAG	04 00	
12097 : 03 :	PCD	93 20	
64 : 04 :	TAG	2a 69 8d 43 8d	
16305 : 05 :	PCD	93 70 2a 69 8d 43 8d 52 55	
64 : 06 :	TAG	08 b6 dd	
16504 : 07 :	PCD	60 04 d1 3d	} Authentication
112 : 08 :	TAG	3b ae 03 2d	
6952 : 09 :	PCD	c4! 94 a1 d2 6e! 96 86! 42	
64 : 10 :	TAG	84 66! 05! 9e!	
396196 : 11 :	PCD	a0 61! d3! e3	} Increment & Transfer
208 : 12 :	TAG	0d	
8442 : 13 :	PCD	26 42 ea 1d f1! 68!	
5120 : 14 :	PCD	8d! ca cd ea	
2816 : 15 :	TAG	06!	
1349238 : 16 :	PCD	2a 2b 17 97	} Read
72 : 17 :	TAG	49! 09! 3b! 4e! 9e! 5e b0 06 d0!	
		07! 1a! 4a! b4! 5c b0! 4f c8! a4!	

Fig. 5. Trace of a card with default keys, recorded by the Proxmark III

reader sends 93 70 followed by the UID and two CRC bytes (#5) to select the card.

Authentication. The card is in the active state and ready to handle any higher layer commands. In Section 2.4 we discussed the authentication protocol. In Figure 5, messages #7 to #10 correspond to the authentication.

The authentication request of the reader is 60 04 d1 3d (#07). The first byte 60 stands for an authentication request with key A. For authentication with key B, the first byte must be 61. The second byte indicates that the reader wants to authenticate for block 4. Note that block 4 is part of sector 1 and therefore this is an authentication request for sector 1. The last two bytes are CRC bytes.

Encrypted Communication. After this successful authentication the card is ready to handle commands for sector 1. The structure of the commands can be recognized clearly. Since we control the MIFARE Classic reader we knew which commands were sent. Message #11 to #15 show how an *increment* is performed. The *increment* is immediately followed by a *read* command (#16 and #17).

5 Weakness in MIFARE Classic

Nohl and Plötz partially recovered the CRYPTO1 algorithm that is used to encrypt the communication between the card and the reader [75]. The pseudo-random generator on the card, which initiates the algorithm by generating a nonce, is weak. In our analysis, we use this weakness to extend the work of Nohl and Plötz with a practical attack, which delivers the needed known plaintext for

brute-force, and a description of the MIFARE Classic protocol. In this attack, we do not need knowledge about the CRYPTO1 algorithm other than that it is a stream cipher which encrypts bitwise.

During our experiments, independently, we also noted the weakness of the pseudo-random generator of the card by requesting many card nonces. We were able to request about 600,000 nonces every hour. Within one hour, a nonce reappeared at least about four times. The nonce is generated by a *Linear Feedback Shift Register* (LFSR) [5] which shifts every 9.44 μs . This is exactly one bit period in the communication. Therefore a random nonce could theoretically reappear after 0.618s, if the card is queried at exactly the right time.

In another experiment, we tried to request a nonce at a fixed time after powering-up [6] the card. This way, we could reduce the card nonces to ten different ones, which decreases the waiting time.

Without knowing the cryptographic algorithm, only an online brute force attack on the key can be mounted. Because of the communication delay, this would take 5ms for each attempt. An exhaustive key search would then take 16,289,061 days, which equals about 44,627 years.

When the cryptographic algorithm is known, an off-line brute force attack can be mounted using a few eavesdropped traces of an authentication run. Nohl and Plötz state that with dedicated hardware of around \$17,000 this would take about one hour. For this attack to work, some known plaintext is required. Our analysis provides this plaintext.

6 Keystream Recovery Attack

In Section 5 we discussed a weakness in the pseudo-random generator of the MIFARE Classic. In this section we deploy a method to recover the keystream that was used in an earlier recorded transaction between a reader and a card. As a result the keystream of the communication will be recovered. For this attack we need to be in possession of the card. The following reasons make this attack interesting:

1. Our attack provides the known plaintext necessary to mount a brute force attack on the key.
2. Using our attack we recovered details about the byte commands.
3. Using the recovered keystream we can *read* card contents without knowing the key.
4. Using the recovered keystream we can also *modify* the contents of the card without knowing the key.

6.1 Keystream Recovery

To recover the keystream we exploit the weakness of the pseudo-random generator. As it is this random nonce in combination with only one valid response of

⁹ As was suggested by Nohl and Plötz [7].

the reader what determines the remaining keystream. For this attack we need complete control over the reader (Proxmark) and access to a (genuine) card. The attack consists of the following steps:

1. Eavesdrop the communication between a reader and a card. This can be for example in an access control system or public transport system.
2. Make sure that the card will use the same keystream as in the recorded communication. This is possible because the card repeats the same nonce in reasonable time, and we completely control the reader.
3. Modify the plaintext, such that the card receives a command for which we know plaintext in the response (e.g., by changing the block number in a read command).
4. For each segment of known plaintext, compute the corresponding keystream segment.
5. Use this keystream to partially decrypt the trace obtained in 1.
6. Try recovering more keystream bits by shifting commands.

The plaintext P_1 in the communication is XOR-ed bitwise with a keystream K which gives the encrypted data C_1 . When it is possible to use the same keystream on a different plaintext P_2 and either P_1 or P_2 is known, then both P_1 and P_2 are revealed.

$$\left. \begin{array}{l} P_1 \oplus K = C_1 \\ P_2 \oplus K = C_2 \end{array} \right\} C_1 \oplus C_2 \Rightarrow P_1 \oplus P_2 \oplus K \oplus K \Rightarrow P_1 \oplus P_2 \quad (1)$$

The weak pseudo-random generator makes it possible to replay an earlier recorded transaction. We can flip ciphertext bits to try to modify the first command such that it gives another result. Another result gives us another plain text. The attack is based on this principle.

6.2 Keystream Mapping

The data is encrypted bitwise. When the reader sends or receives a message, the keystream is shifted the number of bits in this message on both the reader and card side. This is needed to stay synchronized and use the same keystream bits to encrypt and decrypt. The stream cipher does not use any feedback mechanism. Despite that, when we tried to reveal the contents of a message sequence using a known keystream of an earlier trace, something went wrong. We recorded an *increment* followed by a *transfer* command. We used this trace to apply our attack and changed the first command to a *read* command which consists of 4 command bytes and delivers 18 response bytes. Together with the parity bits this makes it a 198 bit stream. The plaintext was known and therefore we recovered 198 keystream bits.

When we used this keystream to map it on the original trace of the *increment* (Figure 6), it turned out that the keystream was not in phase after the first command. The reason was the short 4-bit answer of the card that is not followed

by a parity bit. In our original trace we are now half way the first response byte. This means that after 4 more bits we arrive at the parity bit in the original trace. However, in our new trace we are then half way the next command byte. To correct this we needed to throw away the keystream bit that was originally used to encrypt the parity bit.

But what to do when we need to decrypt a parity bit in the new situation and we are half way a byte with respect to the first trace? The solution is to encrypt the parity bit with the next bit from the recovered keystream and use this same keystream bit to decrypt the next data bit.

From this we can conclude that parity bits are encrypted with keystream bits that are also used to encrypt databits.

	INCREMENT	ACK	VALUE	TRANSFER	ACK
Plaintext	c1 04 f6 8b	0a	01 00 00 00 bb 4a	b0 04 ea 62	0a
Ciphertext	4c 88 31 bc!	0a!	e2 79!2a!14 35!6f!	04!81 2d!1e!	0c!

Fig. 6. Recovering the Keystream and Commands

The following method successfully maps the keystream on another message sequence as we described above.

Take the recovered keystream and strip all the keystream bits from it that were at parity bit positions. The remaining keystream can be used to encrypt new messages. Every time a parity bit needs to be encrypted, use the next keystream bit without shifting the keystream, in all other cases use the next keystream bit and shift the keystream.

6.3 Authentication Replay

To replay an authentication we first need a trace of a successful authentication between a genuine MIFARE reader and card. An example of an authentication followed by one read command is shown below.

```

1 PCD 60 03 6e 49
2 TAG e0 92 93 98
3 PCD ad e7 96! 48! 20! 22 df 93
4 TAG bf 06 91! 82
5 PCD b5! 05! 47 3f
6 TAG 3f 14! 4f e9! 86 38! 96! 85 3e!
   f3 e3! 3d! eb! 2b! a2 d4 dd 76!
```

After we recorded an authentication between card and reader, we do not modify the memory. This ensures that the memory of the card remains unaltered and therefore it will return the same plaintext. Now we will act like a MIFARE reader and try to initiate the same authentication. In short:

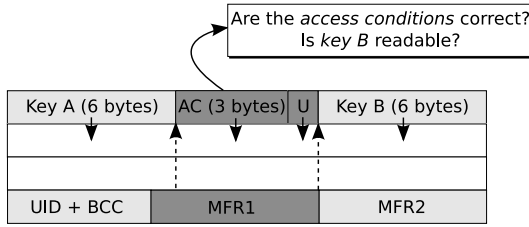


Fig. 7. Recovering Sector Zero

1. We recorded a trace of a successful authentication between a genuine card and reader.
2. We send authentication requests (#1) until we get a nonce that is equal to the one (#2) in the original trace.
3. We send the recorded response (#3) to this nonce. It consists of a valid response to the challenge nonce and a challenge from the reader.
4. We retrieve the response (#4) to the challenge from the card.
5. Now we are at the point where we could resend the same command (#5) or attempt to modify it.

6.4 Reading Sector Zero

We will show that it is possible to read sector 0 from a card without knowing the key. We only need one transaction between a genuine MIFARE reader and card. Every MIFARE Classic card has some known memory contents. The product information published by NXP [8] gives this information.

When a sector trailer is read the card will return logical ‘0’s instead of key A because key A is not readable. If key B is not readable the card also returns logical ‘0’s. It depends on the access conditions if key B is readable or not. The access conditions can be recovered by using the manufacturer data. Block 0 contains the UID and BCC followed by the manufacturer data. The UID and BCC cover 5 bytes and are known. The remaining 11 bytes are covered by the manufacturer data. Some investigation on different cards (MIFARE Classic 1k and 4k) revealed that the first 5 bytes of the manufacturer data almost never change. These bytes (MFR1) cover the positions of the access conditions (AC) and the unknown byte U, as shown in Figure 7. This means that the keystream can be recovered using the known MFR1 bytes by reading block 0 and block 3 (sector trailer) subsequently. Remember that the access conditions are stored twice in 3 bytes. Once inverted and once non-inverted. This way it is easy to detect if we indeed revealed the access conditions. The unknown byte U can be in any state when the card leaves the manufacturer but appears to be often 00 or 69.

The access conditions tell us whether key B is readable or not. In many cases key B is not readable, for instance as in the OV-Chipkaart¹⁰ that is used

¹⁰ MIFARE Classic 4k card.

in the Dutch public transport system. The first 5 bytes of the manufacturer data (MFR1 in Figure 7) recovered the access conditions for sector 0. Because the access conditions for the sector trailer define key B as not readable, we know the plaintext is zeros. Hence the whole sector trailer was revealed and therefore the contents of the whole sector 0 were revealed as well.

7 Reading Higher Sectors

In the higher sectors of the MIFARE Classic card we do not have the advance of the manufacturer data. We basically have the sector trailer and some unknown data blocks. Because of key A we can recover always the first 10 keystream bytes. Key B is in most cases not readable and therefore will give 6 more keystream bytes, but leaves us with a gap of 4 bytes (AC and U).

Although it is harder to achieve, there is a potential threat for these sectors to become compromised.

7.1 Proprietary Command Codes

At the time this research was performed, we were not aware that the command codes, which we revealed with our attack, could already be found in example firmware of NXP¹¹. Note that the firmware refers to the command codes sent from PC to reader. Our research shows that (perhaps obviously) these are the same command codes sent from reader to card.

We used a card in transport configuration with default keys and empty data blocks to reveal the encrypted commands used in the high-level protocol. All the commands send by the reader consist of a command byte, parameter byte and two CRC bytes. We made several attempts to reveal the command by modifying the ciphertext of this command. The way to do this is to assume we actually know the command. With this ‘knowledge’ we XOR the ciphertext which gives us the keystream. To check if this is indeed the correct keystream, we XOR it with a new command for which we know the response. If we guessed the initial command right the response of the card will be that known response. This method revealed the commands shown in Figure 8.

Now, one could try to replay the same authentication again and try to execute a command that returns an ACK or NACK in order to recover more keystream. Because an ACK or NACK is only 4 bits in size, it leaves some spare bits for which we know the keystream. We can use these bits to execute another command for which we now know the plaintext. This delivers more known keystream as a result, and this method can be applied repeatedly. However, this approach does only work if a *decrement*, *increment* or *transfer* is allowed. These are the commands that return an ACK and therefore are in total shorter than the *read*. We can only send valid commands because otherwise the protocol aborts. The *read* command returns 16 data bytes and 2 CRC bytes. On a *write* command

¹¹ <http://www.nxp.com/files/markets/identification/download/MC081380.zip>

Authentication					
READER	CARD	READER	CARD		
60 YY*	Using KeyA	4-byte nonce	8-byte response		
61 YY*	Using KeyB	4-byte nonce	8-byte response		
Data					
READER	CARD	READER			
30 YY*	Read	16 data bytes*			
A0 YY*	Write	ACK / NACK	16 data bytes*		
Value blocks					
READER	CARD	READER	READER		
C0 YY*	Decrement	ACK / NACK	4-byte value*		
C1 YY*	Increment	ACK / NACK	4-byte value*		
C2 YY*	Restore	ACK / NACK	4-byte value*		
B0 YY*	Transfer	ACK / NACK	Transfer		
Other					
READER					
50 00*	Halt				
<table border="1"> <tr> <td>YY = block address</td> </tr> <tr> <td>* = Followed by two CRC bytes</td> </tr> </table>				YY = block address	* = Followed by two CRC bytes
YY = block address					
* = Followed by two CRC bytes					
Card responses (ACK / NACK)					
A (1010)	ACK				
4 (0100)	NACK, not allowed				
5 (0101)	NACK, transmission error				

Fig. 8. Command set of MIFARE Classic

the card returns a 4-bit ACK, this indicates that the card is ready to receive 16 data bytes followed by 2 CRC bytes.

The *decrement*, *increment* and *restore* commands all follow the same procedure. The card indicates that it is expecting a value from the reader by sending a 4-bit ACK response. This value is 4 bytes and is followed by 2 CRC bytes. For the *restore* this value is send but not used. The value is send as YY YY YY YY ZZ ZZ, where YY are the value bytes and ZZ the CRC bytes.

Finally, a *transfer* command is send to transfer the result of one of the previous commands to a memory block. The card response is an ACK if it went well. Otherwise it responds with a NACK.

The 4-bit ACK is 0xa. When a command is not allowed the card sends 0x4. When a transmission error is detected the card sends 0x5. The card does not even give a response at all if the command is of the wrong length. The protocol aborts on every mistake or disallowed command.

8 Conclusions and Recommendations

We have implemented a successful attack to recover the keystream of an earlier recorded transaction between a genuine MIFARE Classic reader and card.

We used a MIFARE Classic reader in combination with a ‘blank’ card with default keys to recover the byte commands that are used in the proprietary protocol. Knowing the byte commands and a sufficiently long keystream allowed us to perform any operation as if we were in possession of the secret key.

We managed to read *all* memory blocks of the sector zero of the card, without having access to the secret key. In general, we were able to read *any* sector of

the memory of the card, provided that we know *one* memory block within this sector. Moreover, after recording a valid transaction on any sector, we were able to read the first 6 bytes of any block in that sector and also the last 6 bytes if key B is read only. Similarly, we are able to *modify* the information stored in a particular sector.

Consequences. First of all, all data stored on the card (except the keys themselves) should no longer be considered secret. In particular, if the MIFARE Classic card is used to store personal information (like name, date of birth, or travel information), this constitutes a direct privacy risk. The security risk is relatively low because in general the security is guaranteed by the secrecy of the keys. Note that in particular we are not able to clone cards, because the secret keys remain secret.

Secondly, the integrity and authenticity of the data stored on the card can no longer be relied on. This is quite a severe security risk. This is particularly worrying in applications where the card is used to store a certain value, like loyalty points or, even worse, some form of digital currency. The loyalty level or the value stored in the electronic purse could easily be increased (or decreased, in a denial-of-service type of attack).

Thirdly, knowledge of the plaintext (or the keystream) is a necessary condition to perform brute force (or other more sophisticated) attacks to recover the secret key. We are making good progress in developing a very efficient attack to recover arbitrary sector keys of a MIFARE Classic card.

Recommendations. For short term improvements we recommend not to use sector zero to store secret information. Configure key B as readable and store random information in it. Do not store sensitive information in the first 6 bytes of any sector. Use multiple sector authentications in one transaction to thwart attackers in an attempt to recover plaintext. This is only helpful when value block commands are not allowed. Value block commands are shorter than a read command and will enable a shift of the keystream. Another possibility, that might be viable for some applications, is to employ another encryption scheme like AES in the backoffice, and store only encrypted information on the tags. To prevent unauthorized modification of a data block, an extra authentication on this data could be added. This authentication is then verified in the backoffice.

Proper fraud detection mechanisms and extra security features in the backoffice are necessary to signal or even prevent the types of attacks described above. In general, the backoffice systems collecting and processing data that comes from the readers are a very important second line of defence.

On the long term these countermeasures will not be sufficient. The MIFARE Classic card has a closed design. Security by obscurity has shown several times that at some point the details of the system will be revealed compromising security [6]. Therefore we recommend to migrate to more advanced cards with an open design architecture.

References

1. Finkenzeller, K.: RFID Handbook, 2nd edn. John Wiley and Sons, Chichester (2003)
2. Garcia, F.D., de Koning Gans, G., Muijrsers, R., van Rossum, P., Verdult, R., Schreur, R.W.: Dismantling MIFARE Classic (forthcoming)
3. Hoepman, J.-H., Hubbers, E., Jacobs, B., Oostdijk, M., Schreur, R.W.: Crossing Borders: Security and Privacy Issues of the European e-Passport. In: Yoshiura, H., Sakurai, K., Rannenberg, K., Murayama, Y., Kawamura, S. (eds.) IWSEC 2006. LNCS, vol. 4266, pp. 152–167. Springer, Heidelberg (2006)
4. ISO/IEC 14443. Identification cards - Contactless integrated circuit(s) cards - Proximity cards (2001)
5. Nohl, S.K., Evans, D., Plötz, H.: Reverse-Engineering a Cryptographic RFID Tag. In: USENIX Security Symposium, San Jose, CA, 31 July (2008)
6. Kerckhoffs, A.: La cryptographie militaire. *Journal des sciences militaires*, IX, pp. 5–38, January 1983, and pp. 161–191, February 1983(1983)
7. Nohl, K., Plötz, H.: MIFARE, Little Security, Despite Obscurity. In: Presentation on the 24th Congress of the Chaos Computer Club in Berlin (December 2007)
8. NXP Semiconductors. MIFARE Standard 4KByte Card IC functional specification (February 2007)

A Chemical Memory Snapshot

Jörn-Marc Schmidt^{1,2}

¹ Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria

`joern-marc.schmidt@iaik.at`

² Secure Business Austria (SBA),
Favoritenstraße 16, 1040 Vienna, Austria

Abstract. Smart cards and embedded systems are part of everyday life. A lot of them contain sensitive data like keys used in secure applications. These keys have to be transferred from non-volatile to static memory to generate signatures or encrypt data. Hence, the possibility to read out the static memory of a device is a crucial security threat. This paper presents a new technique to read out secret data from the internal static memory of a cryptographic device. A chemical reaction of the top metal layer of a decapsulated chip is used to identify lines connected to the positive power supply. Using this information, we are able to obtain the content of memory cells like the secret key of a cryptographic system.

Keywords: Smart cards, physical security, electrolysis.

1 Introduction

Evaluating the security of a cryptographic device, not only the used protocol and its underlying cryptographic algorithms are important. The device itself and the way the algorithms are implemented on it may also reveal valuable information. Attacks that exploit properties of the device are called implementation attacks. Depending on whether the behavior of a device is influenced or just measured, an attack is called active or passive. Furthermore, implementation attacks can be non-invasive, semi-invasive, or invasive. Non-invasive attacks do not modify the package of the device, while semi-invasive attacks apply a decapsulation procedure to expose the chip. In addition, if direct electrical contact is established to the surface of the chip, the attack is called invasive.

Non-invasive, passive ones are called side-channel attacks [1]. Paul Kocher utilized differences in the execution time depending on secret data to reveal it in 1996 [2]. In 1997, it has been shown by Eli Biham and Adi Shamir [3] and in parallel by Dan Boneh et al. [4] that actively provoking faults in cryptographic devices can also be exploit to uncover secret information. That passive measurement of the power consumption of a device may also reveal secret data was demonstrated by Paul Kocher et al. in 1999. Subsequently, it has been shown by Dakshi Agrawal et al. in 2002 that measuring electromagnetic emissions of a device allows to attack it. Side channel as well as fault attacks have become

a popular topic in research. Mostly, multiple calculations of encryptions or signatures, a previous characterization of the device, or a precise fault injection are necessary for an attack. Another way to disclose a secret key is reading it directly out of the memory of a device, which can be done without using the reading operations of the device. This method is independent of the implemented algorithm as long as the key is processed inside the static memory.

A common way to read out secret data directly is probing [5,6]. Thereby, a probing needle establishes electrical contact to the surface of the chip. As the technology size is getting smaller, this becomes more and more difficult.

Another way to reveal content of memory cells was presented by David Samyde et al. [7]. They scanned the surface of an active chip with a laser beam. At each position of the laser, the current injected by the beam was measured. In this way, cells containing zeros could be distinguished from those containing ones. Their method is semi-invasive, as a decapsulation procedure has to be applied. As several points have to be scanned, the technique is rather slow. To read out important data during a computation that cannot be stopped in the target state, they suggest freezing the memory to increase the remanence of the data in the cells [8].

Our Contribution. We present a new method to produce a quick one-time snapshot of the state the memory cells are in. Our method makes use of a chemical reaction called electrolysis. For this purpose, the chip has to be decapsulated and parts of the passivation must be removed. The reaction shows the current state of the exposed memory cells. A standard procedure for decapsulation as described in [9] is sufficient. Exposing the top metal layer can be done with different procedures. Wet etching for removing the whole passivation or a laser cutter as well as a focused ion beam (FIB) for a selective removal can be used. We will elaborate on these possibilities in Section 4. For our experiments we decided to use a laser cutter, as it is the cheapest choice.

This paper is organized as follows. After describing the target device including details on SRAM cells in Section 2, the chemical process called electrolysis is explained in Section 3. Possible ways to remove the passivation of a chip are discussed in Section 4. Section 5 shows the results of our experiments. Conclusion is drawn in Section 6.

2 Target Device

An 8-bit microcontroller with 128 byte internal static memory was chosen for the experiments. Its passivation consists of silicon-dioxide, the top metal layer of aluminum with a titanium-nitride barrier. The size of a memory cell in the device is $474 \mu\text{m}^2$. In the following, its structure will be explained in more detail.

A standard static memory cell consists of six transistors. Four of them build two inverters. Each output of the invertors is connected to the input of the other one. Read and write functions of the cell are realized by the two remaining transistors. Figure 1 shows a schematic of a standard cell. For programming a cell the appropriate value is put onto the bit line, its inverse onto the inverse bit line. Programming is enabled by the select line. As the drivers of the write

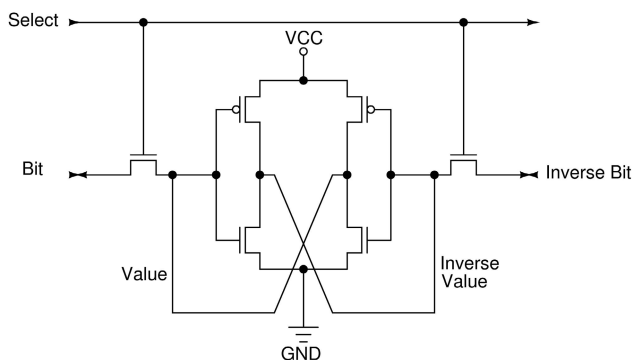


Fig. 1. Schematic of a standard SRAM cell

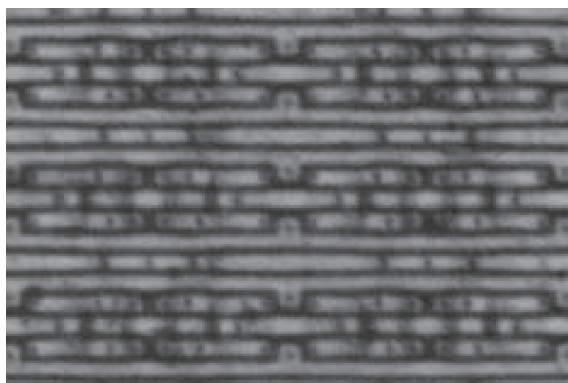


Fig. 2. Array of SRAM cells (metal layer)

circuit are stronger than the transistors inside the cell, the old state is replaced by the one put onto the bit lines. Sense amplifiers are used to read a cell. They recognize the state of the cell activated by the select line.

If the cell contains one, the value line is connected to the positive power supply (VCC) and the inverse value line to ground (GND); vice versa if it contains zero. Figure 2 shows the top metal layer of three times four memory cells. In the chip considered, parts of the circuit lines containing the value and the inverse value of the cell are realized on the top metal layer. Thus, these two lines indicate whether a cell contains zero or one. If the metal layer is exposed, this information can be gained by electrolysis.

3 Electrolysis

Electrolysis is a chemical process. Thereby, electrical energy is converted into chemical energy in liquids containing ions. Such a liquid is called electrolyte.

The electrical energy has to be supplied as direct current to it. The supplying conductors are called electrodes. Electric charge in electrolytes is carried by its ions. These ions move within the electrolyte and cause chemical reactions at the electrodes.

In an electrolytic process the electrode that emits electrons is called cathode. Its opposite is called anode. At anode and cathode different chemical reactions take place: at the cathode positive charged ions, named cations, are reduced; at the anode negative charged ions, named anions, are oxidized. Thus, the conduction of an electrolyte depends on the mobility of its ions. The electrolyte itself stays electrical neutral [10]. Industrial processes commonly use electrolyze for separating chemicals, as well as for putting a protective layer on materials.

Here, electrolysis is applied for attacking a device. As the distance between the metal lines, which will act as anode and cathode, is very small, a liquid that conducts only sparely is necessary. This reduces the damage caused by the current flowing over the liquid. Pure water is only sparely conducting. Therefore, distilled water from the local tool store was used. The conductivity of tap water is much higher, because it contains dissolved salts and thus free ions. For electrolyze to take place on the chip, it is necessary that the liquid has direct contact to the surface of the top metal layer. Thus, the passivation of the chip has to be removed, at least from the memory cells of interest.

4 Removal of the Passivation

Removing the passivation is a quite more challenging task than the package decapsulation. The passivation of a common chip consists of silicon oxide, often in combination with a layer of silicon nitride. Those layers can be removed from the whole chip at once or in a selective way, which exposes only small parts of the chip.

The whole passivation can be removed at once by wet etching [11]. This process always acts uniform in all directions. A silicon nitride layer can be removed by 85 % phosphoric acid at 160 °C. Silicon oxide can only be etched with hydrogen fluoride. In order to be able to stop the reaction before underlying layers are affected, a buffered dissolution of hydrofluoric acid and ammonium fluoride is commonly used. There are several different mixtures. It is necessary to know the etch rate of the used mixture as well as the thickness of the silicon oxide layer to stop the reaction at the right moment.

A selective removal of the layers can be performed by a focused ion beam (FIB) or a laser cutter. Focused ion beams work similar to scanning electron microscopes (SEMs). The electron beam in the electron microscope is substituted by a beam of gallium ions. At low power this beam can be used for imaging, at higher power for milling. Using the ion beam a milling with a precision of submicron scale can be achieved. In contrast to a focused ion beam, a laser cutter emits a pulsed light beam. This beam is focused by a microscope. Depending on its intensity, the beam can expose or at higher optical output cut wires of the top metal layer.

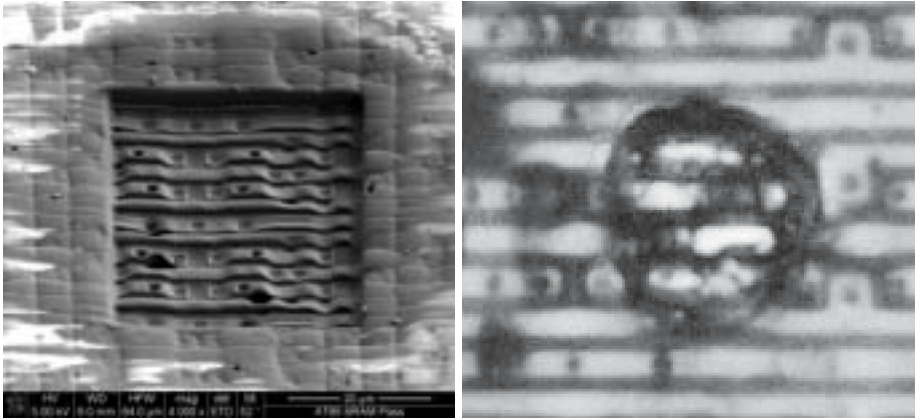


Fig. 3. Passivation removal using a focused ion beam (left) and a laser cutter (right)

While a focused ion beam can remove the passivation in a very careful and well directed way, a laser cutter needs a very precise adjustment. A strong beam can damage the circuit and weak beam may have no effect at all. In Figure 3 the differences between removing the passivation with a focused ion beam and laser cutter are shown. For the experiments a laser cutter was utilized, because of the high cost of a focused ion beam. One shot of the laser cutter exposed parts of the bit and the inverse bit lines. This was sufficient for the attack.

5 Results

With a small pipette a water drop of distilled water was put onto the surface of the powered chip. In order to avoid unwanted inferences, the water should not get in touch with the bonding wires. Using this setup, we were still able to write and read the values of the cells for several minutes with the program of the microcontroller. Afterwards, errors occurred in some memory cells.

Immediately after the water reaches the surface, the chemical reaction begins. This is indicated by a liberated gas. Before and during the experiment, the value one was written to a memory cell. Hence, its value was not changed. Figure 4 shows the result of the reaction. The line containing the bit and the positive programming line have been stained, even parts underneath the passivation, while the inverse lines did not change their color. The ground line was hit by the laser cutter without cutting it completely. The chemical process had no influence on it.

If the value is not changed during the procedure, the staining does not change, even if the metal lines are exposed to the water for just a few seconds or several minutes. At cells that change their value while they are in touch with the water, each of the two value lines act as anode and as cathode. Thus, both lines show the same staining. Considering a computation, it is possible to distinguish between memory cells where data is processed and unused or cells with static values. The

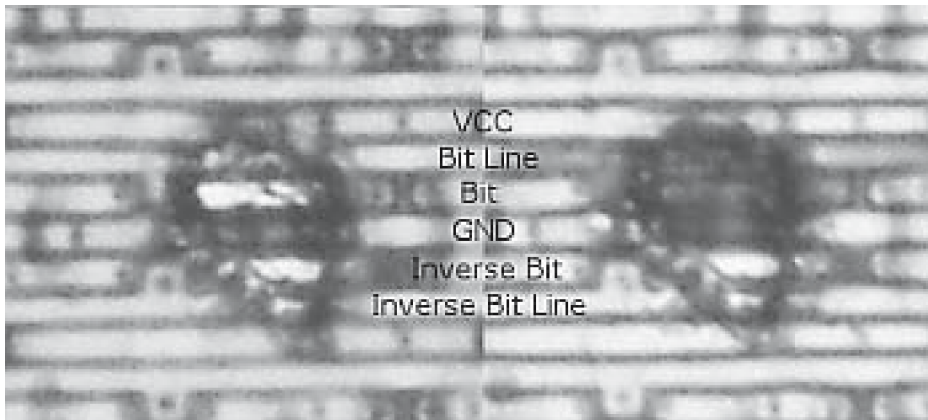


Fig. 4. Memory cell before (left) and after chemical preparation (right)

change of the color is an irreversible process. Once the color of a line has been changed, we were not able to remove the staining.

6 Conclusion

We presented a new way to read out memory without using the functions supplied by the chip. The method makes use of a chemical reaction. It produces a one-time snapshot of the actual state of the memory. Thus, values processed in the static memory of a device can be read out by an attacker, including secret keys.

Acknowledgments

The author would like to thank Julian Wagner and the Institute for Electron Microscopy of the TU Graz for their support and the focused ion beam picture. I would also like to thank Peter Söser and Christoph Marschner for their support.

References

1. Mangard, S., Oswald, E., Popp, T.: *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, Heidelberg (2007)
2. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) *CRYPTO 1996*. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
3. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) *CRYPTO 1997*. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: Fumy, W. (ed.) *EUROCRYPT 1997*. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)

5. Anderson, R.J., Kuhn, M.G.: Tamper Resistance - a Cautionary Note. In: Second Usenix Workshop on Electronic Commerce, pp. 1–11 (November 1996)
6. Kömmerling, O., Kuhn, M.G.: Design Principles for Tamper-Resistant Smartcard Processors. In: USENIX Workshop on Smartcard Technology (Smartcard 1999), pp. 9–20 (May 1999)
7. Samyde, D., Skorobogatov, S.P., Anderson, R.J., Quisquater, J.J.: On a New Way to Read Data from Memory. In: IEEE Security in Storage Workshop (SISW 2002), pp. 65–69. IEEE Computer Society, Los Alamitos (2002)
8. Skorobogatov, S.: Low temperature data remanence in static RAM. Technical report, University of Cambridge Computer Laboratory (June 2002)
9. Skorobogatov, S.P.: Semi-invasive attacks - A new approach to hardware security analysis. PhD thesis, University of Cambridge - Computer Laboratory (2005), <http://www.cl.cam.ac.uk/TechReports/>
10. Gärtner, H., Hoffmann, M., Schaschke, H., Schürmann, I.M.: Das große Buch der Chemie. Compact Verlag (2004)
11. Beck, F.: Integrated Circuit Failure Analysis: A Guide to Preparation Techniques. Wiley, Chichester (1998)

Recent Advances in Electronic Cash Design

Aline Gouget

Security Labs, Gemalto,
6, rue de la Verrerie, F-92190 Meudon, France
aline.gouget@gemalto.com

Abstract. Electronic cash (or e-cash) is an electronic payment solution that is usually viewed as an attempt to emulate electronically the main characteristics of regular cash. In particular, e-cash and other payment solutions should protect the privacy of users during a purchase. The main distinction of e-cash with respect to other electronic payment systems is that electronic coins are stored on a device controlled by the user, e.g. a smart card or a personal computer hard disk. Since the introduction by Chaum [10,11] of unconditionally untraceable electronic money, e-cash systems have been extensively studied. Recent work has mainly focused on the efficiency of the protocols with respect to several notions of anonymity. In this talk, we will review the main recent results and also discuss the possibility to transfer a coin without involving the bank which is considered as an important characteristic of regular cash.

1 Overview of e-Cash Schemes

E-cash systems usually assume that the same bank is responsible for giving out electronic coins and for later accepting them for deposit. Users can download a number of electronic coins from the bank using a withdrawal protocol, and next pay one or more merchants with them in a spending protocol. Merchants can later exchange electronic coins for regular cash on their bank account using a deposit protocol.

As it is easy to duplicate electronic data, an e-cash system requires a mechanism that prevents a user from spending the same coin twice without being identified, and it must also prevent a merchant from depositing the same coin twice. E-cash systems allow merchants to check the validity of coins, whereas the detection of double spending is performed by the bank. Indeed, double-spending cannot be checked by the merchant during a payment protocol, as the coins delivered by the bank can be spent at several merchants.

E-cash systems can be categorized into two groups according to whether the bank is *on-line* or not in the spending protocol. In on-line e-cash, a merchant only accepts a coin if the bank confirms that the coin has not been previously spent, and the deposit protocol must be performed immediately after the spending protocol. This scenario is often considered as being very restrictive in practice, especially for low-value payments. In off-line e-cash, the merchant does not need to interact with the bank before accepting a coin from the user. Indeed, during

the spending of a coin, the merchant only checks the validity of the coin. Nevertheless, the merchant is guaranteed that the bank will accept the coin or the bank will be able to identify and punish the cheater.

E-cash should provide user anonymity against both the bank and the merchant during a purchase in order to emulate the perceived anonymity of regular cash transaction. When a double-spending is detected, the identity of the cheater must be retrieved. Off-line e-cash schemes can also be categorized into two groups according to whether the revocation of the cheater identity is either done by a trusted party, e.g. a judge, (in this case the revocation of the spender identity is always “technically feasible” by the trusted party), or technically possible only in case of a double-spending.

The main security properties usually considered in e-cash schemes are the unforgeability of coins, the anonymity of users, the unlinkability of spends, the identification of double-spenders and the impossibility for the bank to falsely accuse (with a proof) honest users. Many e-cash schemes have been proposed in the literature, which fulfill some of the security properties previously mentioned, in the on-line or off-line setting, involving a judge or not. Only few of them consider the possibility to transfer a coin from a user to another user without involving the bank.

2 Towards a *Practical* e-Cash Scheme

Most recent work has focused on the efficiency of protocols, i.e. the efficiency of the algorithms executed during a protocol and the compactness of the data exchanged between all actors. A major challenge in e-cash is to provide an efficient solution to spend several coins at the same time, i.e., more efficiently than iterating the spending protocol over each coin”.

The main significant improvement has been done by Camenisch et al. [4] by introducing the compact e-cash scheme that allows a user to withdraw efficiently a wallet containing 2^L coins such that the space required to store these coins and the complexity of the withdrawal protocol are proportional to $(L + k)$ rather than $(k \cdot 2^L)$, where L is a fixed parameter of the system and k is a security parameter. This scheme fulfills the anonymity and unlinkability properties usually required for electronic cash systems. The main drawback of the compact e-cash system is that it does not address the possibility for spending several coins at the same time without iterating the execution of the spending protocol. We will review recent improvements and variants of the compact e-cash scheme that have been proposed [19][7][3][1].

Divisible e-cash schemes attempt to address the problem of the *divisibility of a coin* by allowing a user to withdraw a coin of monetary value 2^L and then to spend this coin in several times by dividing the value of the coin. The aim is to allow a user to spend a coin of monetary value 2^ℓ more efficiently than repeating 2^ℓ times a spending protocol. Many off-line *divisible e-cash* systems have been proposed in the literature (e.g. [17][13][14][16][9][15][5][2]). We will review the main advantages and drawbacks of these schemes.

3 On the Transferability Property in e-Cash

The transferability property of a coin, meaning that received cash can be spent later without involving the bank, is seen as a fundamental property of regular cash. However, it has received only little attention in the electronic setting. This lack of interest for transferable e-cash may be explained by the result given in [12] showing that it is impossible to transfer a coin without increasing its size. However, the main advantage of the transferability of e-cash would be the decrease of the number of communications between the bank and all users. We will review the main advantages and drawbacks of transferable e-cash schemes that have been proposed in the literature [17,18,12,6,8].

References

1. Au, M.H., Susilo, W., Mu, Y.: Practical compact e-cash. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP 2007. LNCS, vol. 4586, pp. 431–445. Springer, Heidelberg (2007)
2. Au, M.H., Susilo, W., Mu, Y.: Practical anonymous divisible e-cash from bounded accumulators. In: PACS 2000. LNCS, vol. 5143. Springer, Heidelberg (2008)
3. Au, M.H., Wu, Q., Susilo, W., Mu, Y.: Compact e-cash from bounded accumulator. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 178–195. Springer, Heidelberg (2006)
4. Camenisch, J., Hohenberger, S., Lysyanskaya, A.: Compact e-cash. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 302–321. Springer, Heidelberg (2005)
5. Canard, S., Gouget, A.: Divisible e-cash systems can be truly anonymous. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 482–497. Springer, Heidelberg (2007)
6. Canard, S., Gouget, A.: Anonymity in Transferable E-cash. In: Bellovin, S.M., Gennaro, R., Keromytis, A., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 207–223. Springer, Heidelberg (2008)
7. Canard, S., Gouget, A., Hufschmitt, E.: A handy multi-coupon system. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 66–81. Springer, Heidelberg (2006)
8. Canard, S., Gouget, A., Traoré, J.: Improvement of Efficiency in (Unconditional) Anonymous Transferable E-Cash. In: PACS 2000. LNCS, vol. 5143. Springer, Heidelberg (2008)
9. Chan, A.H., Frankel, Y., Tsiounis, Y.: Easy come - easy go divisible cash. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 561–575. Springer, Heidelberg (1998)
10. Chaum, D.: Blind signatures for untraceable payments. In: Crypto 1982, pp. 199–203. Plenum Press, Springer (1982)
11. Chaum, D.: Blind signature system. In: Crypto 1983, p. 153. Plenum Press, Springer (1983)
12. Chaum, D., Pedersen, T.P.: Transferred Cash Grows in Size. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 390–407. Springer, Heidelberg (1993)
13. D’Amiano, S., Di Crescenzo, G.: Methodology for digital money based on general cryptographic tools. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 156–170. Springer, Heidelberg (1995)

14. Eng, T., Okamoto, T.: Single-term divisible electronic coins. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 306–319. Springer, Heidelberg (1995)
15. Nakanishi, T., Sugiyama, Y.: Unlinkable divisible electronic cash. In: Okamoto, E., Pieprzyk, J.P., Seberry, J. (eds.) ISW 2000. LNCS, vol. 1975, pp. 121–134. Springer, Heidelberg (2000)
16. Okamoto, T.: An efficient divisible electronic cash scheme. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 438–451. Springer, Heidelberg (1995)
17. Okamoto, T., Ohta, K.: Universal electronic cash. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 324–337. Springer, Heidelberg (1992)
18. van Antwerpen, H.: Electronic Cash. Master's thesis, CWI (1990)
19. Wei, V.K.: More compact e-cash with efficient coin tracing. Cryptology ePrint Archive, Report 2005/411 (2005), <http://eprint.iacr.org/>

Author Index

- Akishita, Toru 206
Almaliotis, Vasilios 17

Bouzefrane, Samia 228
Buchmann, Johannes 104
Burmester, Mike 176

Carbonell, Mildrey 241
Cayrel, Pierre-Louis 191
Cordry, Julien 228
Costan, Victor 133

Dahmen, Erik 104
de Koning Gans, Gerhard 267
de Medeiros, Breno 176
Devadas, Srinivas 133
Dillinger, Oliver 149
Drissi, Mhamed 218
Dubois, Vivien 218
Durand, Alain 255

Eisenbarth, Thomas 104
Éluard, Marc 255

Gaborit, Philippe 191
Garcia, Flavio D. 267
Ghindici, Dorina 32
Gouget, Aline 290
Guilloux, Anne-Marie 218

Hoepman, Jaap-Henk 267
Hofferek, Georg 162

Katagi, Masanobu 206
Katsaros, Panagiotis 17
Kim, Chong Hee 48

Langer, Josef 149
Leander, Gregor 89
Lelievre, Sylvain 255
Loizidis, Alexandros 17
Louridas, Panagiotis 17

Madlmayr, Gerald 149
Malek, Wael William Zakhari 118

Markantonakis, Kostas 118
Mayes, Keith 118
Meunier, Hervé 228
Miyato, Yoshikazu 206
Mizuno, Asami 206
Mostowski, Wojciech 1
Motta, Rossana 176

Nikodem, Maciej 61

Okeya, Katsuyuki 74

Paar, Christof 89, 104
Paradinas, Pierre 228
Poll, Erik 1
Poschmann, Axel 89
Prouff, Emmanuel 191

Quisquater, Jean-Jacques 48

Réal, Denis 218
Rohde, Sebastian 104
Rolfes, Carsten 89

Sarmenta, Luis F.G. 133
Scharinger, Josef 149
Schmidt, Jörn-Marc 283
Shibutani, Kyoji 206
Sierra, Jose M. 241
Simplot-Ryl, Isabelle 32
Spinellis, Diomidis 17

Tellez, Jesus 241
Torres, Joaquin 241

Valette, Frédéric 218
van Dijk, Marten 133
Vincent, Christophe 255
Vuillaume, Camille 74

Wolkerstorfer, Johannes 162

Yoshino, Masayuki 74