Analysis of the IBM CCA Security API Protocols in Maude-NPA*

Antonio González-Burgueño[1], Sonia Santiago[1], Santiago Escobar[1], Catherine Meadows[2], and José Meseguer[3]

[1] DSIC-ELP, Universitat Politècnica de València, Spain
{agonzalez,ssantiago,sescobar}@dsic.upv.es
[2] Naval Research Laboratory, Washington DC, USA
meadows@itd.nrl.navy.mil
[3] University of Illinois at Urbana-Champaign, USA
meseguer@illinois.edu

Abstract. Standards for cryptographic protocols have long been attractive candidates for formal verification. It is important that such standards be correct, and cryptographic protocols are tricky to design and subject to non-intuitive attacks even when the underlying cryptosystems are secure. Thus a number of general-purpose cryptographic protocol analysis tools have been developed and applied to protocol standards. However, there is one class of standards, security application programming interfaces (security APIs), to which few of these tools have been applied. Instead, most work has concentrated on developing special-purpose tools and algorithms for specific classes of security APIs. However, there can be much advantage gained from having general-purpose tools that could be applied to a wide class of problems, including security APIs.

One particular class of APIs that has proven difficult to analyze using general-purpose tools is that involving exclusive-or. In this paper we analyze the IBM 4758 Common Cryptographic Architecture (CCA) protocol using an advanced automated protocol verification tool with full exclusive-or capabilities, the Maude-NPA tool. This is the first time that API protocols have been satisfactorily specified and analyzed in the Maude-NPA, and the first time XOR-based APIs have been specified and analyzed using a general-purpose unbounded session cryptographic protocol verification tool that provides direct support for AC theories. We describe our results and indicate what further research needs to be done to make such protocol analysis generally effective.

Keywords: IBM 4758 Common Cryptographic Architecture, security Application Programming Interfaces (security APIs), symbolic cryptographic protocol analysis, automatic reasoning modulo XOR theory.

* Antonio González-Burgueño, Sonia Santiago and Santiago Escobar have been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN 2010-21062-C02-02 and TIN 2013-45732-C4-1-P, and by Generalitat Valenciana PROMETEO2011/052. José Meseguer has been partially supported by NSF Grant CNS 13-10109.

L. Chen and C. Mitchell (Eds.): SSR 2014, LNCS 8893, pp. 111–130, 2014.

1 Introduction

Standards for cryptographic protocols have long been attractive candidates for formal verification. Cryptographic protocols are tricky to design and subject to non-intuitive attacks even when the underlying cryptosystems are secure. Furthermore, when protocols that are known to be secure are implemented as standards, the modifications that are made during the standardization process may introduce new security flaws. Thus a considerable amount of work has been done in the application of formal methods to cryptographic protocol standards [26,4,25,1]. In this work the protocols are treated *symbolically*, with the cryptosystems treated as black-box function symbols. The formal methods tool attempts to show that there is no way an attacker, by interacting with the protocol and applying the cryptographic functions symbols in any order, can break the security of the protocol. Such tools can be used both to search for attacks and to prove security with respect to the symbolic model.

Such symbolic formal analyses can be of great benefit to standards development. The environment in which these standards must be developed makes it difficult to maintain security. Standards often must compromise between different and conflicting requirements. The main focus is often interoperability instead of security. Moreover, standards are often evolving documents; they do often must be updated as new requirements arise. However, standards are chiefly intended as guides to implementation, and often contain little information about the security decisions that were made in the design of previous versions of the protocol.[1] All of this means that security flaws often creep into a standard even when it is based on a protocol that was originally secure. Symbolic formal analysis can provide a rapid means of evaluating and re-evaluating the security of a standard and the security requirements it must satisfy as it evolves.

Most symbolic analysis work has concentrated on standards for key generation and secure communication, as these are the types of protocols that are most widely standardized. However, recently another type of application has begun to attract interest: secure Application Programming Interface (API) protocols. This is the functionality a secure device provides for use by applications that run on it. The API allows the application to authenticate itself to the device and perform the functions it is authorized to do. However, it must also be constructed so that the application can not use it to perform any actions that it is *not* authorized to do. For example, it should not be able to obtain cryptographic keys in the clear. It is clearly more economical, both from the point of view of guaranteeing security and producing applications, if APIs are standard across different platforms, and as a result such standards as the IETF's GSSAPI [21] have appeared. But even when an API is not standardized across different platforms, but is created by a single company or other entity to guide application implementers in the use of the devices it creates, it still has many of the properties of a standard. The focus of the documentation is more on implementation than explaining security decisions,

[1] The IETFs insistence on a Security Considerations section in every document is an attempt to address this last problem.

and the APIs often evolve as the hardware and the requirements it must satisfy evolve. Moreover, they are widely distributed and available for formal analysis. Thus lessons learned by analysis of APIs that are not official standards can still be useful to the designers of such standards.

APIs face many of the same issues as key distribution protocols. However, although some of the earliest formal cryptographic protocol analysis work was applied to security APIs [19,22,23], it was a long time before there was any work following up on that. Indeed, it was not until more recent work uncovered security problems in a number of well-known APIs, such as Bond's discovery of flaws in the IBM 4758 Common Cryptographic Architecture API (CCA-API) [3] that this again became an active area of investigation. Even so, application of symbolic formal methods tools for cryptographic protocol analysis to this problem have not been that common until recently. Even now, work has mostly concentrated on developing special-purpose algorithms and tools fine-tuned for specific classes of APIs, rather than expanding general-purpose cryptographic protocol analysis tools to deal with this kind of problem. Indeed, even though Bond's attacks on CCA were discovered almost fifteen years ago, and they have become one of the benchmarks for symbolic protocol analysis, general-purpose tools often still struggle with them.

One of the reasons we believe that general-purpose symbolic cryptographic protocol analysis tools have not been applied yet that widely to security APIs is that the analysis of some API protocols involves features that are not usually considered in the analysis of cryptographic protocols. An illustrative example of this case is the work of Mukhamedov *et. al.* [28]. In this work the authors analyze a fragment of the API for a Trusted Platform Module in ProVerif [2], but encountered problems in encoding state information and in handling such information during the analysis. However, we note that the model of APIs and their desired behavior is possible to formalize and verify by hand, as in [5,10]; the issue here is implementing the appropriate functionality into cryptographic protocol analysis tools.

Another reason that is perhaps harder to address is that many of the APIs rely on properties of the cryptoalgorithm that are not supported by many of the tools, or are supported only partially. Many of these properties can be expressed as equations describing the behavior of the crypto system. For example, CCA-API makes extensive use of exclusive-or, which satisfies the following equations, where $*$ denotes the exclusive-or symbol:

$$\begin{aligned} x * (y * z) &= (x * y) * z && \text{(associativity)} \\ x * y &= y * x && \text{(commutativity)} \\ x * 0 &= x && \text{(neutral element)} \\ x * x &= 0 && \text{(self-cancellation)} \end{aligned}$$

Although there are a number of tools, e.g. ProVerif, that can deal with equations that can be expressed as rewrite rules (that is, that can be given an orientation), tools that can deal with equations that involve both associativity and commutativity (AC) are rarer. Even those tools that do support AC theories do not always support exclusive-or. For example, the Tamarin tool is optimized [27] for modular exponentiation and bilinear pairing, but has not been applied to or optimized for exclusive-or. However, the problem is not necessarily completely intractable. Hand proofs have been developed for some APIs, a number of decision procedures have been developed for the bounded session model, in which the attacker can interact with the protocol only a finite number of times [6,7], and others have been developed for the unbounded session model for certain subclasses of protocols [8,32,9]. In particular, the class of algorithms addressed by [9] is focused on IBM-CCA-like protocols, and has been applied to several versions of IBM-CCA, including the ones analyzed in this paper. Steel [31] has also proposed the use of *XOR constraints* and applied them to the analysis of the IBM CCA protocols, as well as some key exchange protocols using XOR, such as a modified version of Needham-Schroeder. However, this also assumes a bounded session model, e.g. a bounded number of executions of the API operators.

There have, however, been some notable exceptions to this rule, in which general cryptographic protocol analysis tools that allow search in the unbounded session model have been applied to protocols using exclusive-or. One is the Maude-NPA protocol analysis tool [12], which supports equational theories having *finite variant decompositions*, which includes exclusive-or. It has been used successfully to analyze a number of protocols that use exclusive-or, e.g. in [13,15], but had not been applied to cryptographic APIs until now. The other is the work of Küsters and Truderung [20], who give an algorithm for compiling a class of xor-based protocols called *XOR-linear* to protocols that can be analyzed via ProVerif, a tool that does not in itself support AC theories. Not all XOR-based protocols are XOR-linear, but in some cases it is possible to transform a protocol to an XOR-linear protocol that is equivalent to the original with respect to secrecy properties. Küsters and Truderung perform such a transformation for the IBM CCA, and then use their algorithm to analyze it in ProVerif.

In this paper we apply Maude-NPA to the analysis of IBM CCA. We analyze not only the original protocol, but the different fixes provided by IBM in [16], and the different XOR-linear versions provided by Küesters and Truderung in [20]. In particular, we seek to reproduce Bond's attack on the different versions. We demonstrate that it is indeed possible to perform analyses of APIs using XOR, and in some cases to achieve termination. We also discuss what needs to be done to improve Maude-NPA's performance.

In addition we demonstrate the use of *never patterns* to refine and guide our search. Maude-NPA finds attacks by searching backwards from an *attack pattern.* A never pattern is a state pattern that can be added to the attack pattern to reduce the size of the search space; if Maude-NPA creates a state in the search tree that contains an instantiation of a never pattern, then it does not look for any children of that state. We show how never patterns can be used in a way that reduces the size of the search space but is *complete* with respect to reachability of the original attack pattern. In some cases it may not be possible to maintain completeness; but we show how never patterns can be used in a way that maintains completeness *with respect to the existence of a particular attack trace or class of traces*. Both types of never patterns were used in the IBM-CCA analyses.

2 Maude-NPA

In this section we give a high-level summary of Maude-NPA, with particular interest paid to the use of never patterns. For further information, please see [12].

2.1 Preliminaries on Unification and Narrowing

We assume an order-sorted signature $\mathbf{\Sigma} = (\mathsf{S}, \leq, \Sigma)$ with a poset of sorts $(\mathsf{S}, \leq)$ and an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s}\in\mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted term algebras. For a term t, $\mathcal{V}ar(t)$ denotes the set of variables in t.

Positions are represented by sequences of natural numbers denoting an access path in the term when viewed as a tree. The top or root position is denoted by the empty sequence ϵ. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the term t where $t|_p$ is replaced by u.

A *substitution* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$.

A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. Σ and a set E of Σ-equations, The E-equivalence class of a term t is denoted by $[t]_E$ and $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/E}$ denote the corresponding order-sorted term algebras modulo E. An *equational theory* (Σ, E) is a pair with Σ an order-sorted signature and E a set of Σ-equations.

An *E-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_E t'\sigma$. For $\mathcal{V}ar(t) \cup \mathcal{V}ar(t') \subseteq W$, a set of substitutions $CSU_E^W(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo E away from W iff: (i) each $\sigma \in CSU_E^W(t = t')$ is an E-unifier of $t = t'$; (ii) for any E-unifier ρ of $t = t'$ there is a $\sigma \in CSU_E^W(t = t')$ such that $\sigma|_W \sqsupseteq_E \rho|_W$ (i.e., there is a substitution η such that $(\sigma \circ \eta)|_W =_E \rho|_W$); and (iii) for all $\sigma \in CSU_E^W(t = t')$, $Dom(\sigma) \subseteq (\mathcal{V}ar(t) \cup \mathcal{V}ar(t'))$ and $Ran(\sigma) \cap W = \emptyset$.

A *rewrite rule* is an oriented pair $l \to r$, where$l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple (Σ, E, R) with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. The (R, E) rewriting relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as: $t \to_{p,R,E} t'$ iff there exist $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_E l\sigma$ and $t' = t[r\sigma]_p$.

Let t be a term and W be a set of variables such that $\mathcal{V}ar(t) \subseteq W$, the R,E-*narrowing* relation on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is defined as $t \rightsquigarrow_{p,\sigma,R,E} t'$ if there is a non-variable position $p \in Pos_{\Sigma}(t)$, a rule $l \to r \in R$ properly renamed s.t. $(\mathcal{V}ar(l) \cup \mathcal{V}ar(r)) \cap W = \emptyset$, and a unifier $\sigma \in CSU_E^{W'}(t|_p = l)$ for $W' = W \cup \mathcal{V}ar(l)$, such that $t' = (t[r]_p)\sigma$.

2.2 Maude-NPA Syntax and Semantics

Given a protocol $\mathcal{P}$, states are modeled as elements of an initial algebra $\mathcal{T}_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$, where $\Sigma_{\mathcal{P}}$ is the signature defining the sorts and function symbols (for the cryptographic functions and for all the state constructor symbols) and $E_{\mathcal{P}}$ is a set of equations specifying the *algebraic properties* of the cryptographic functions and the state constructors. Therefore, a state is an $E_{\mathcal{P}}$-equivalence class $[t]_E \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$ with t a ground $\Sigma_{\mathcal{P}}$-term.

In Maude-NPA a *state pattern* for a protocol P is a term t of sort State (i.e., $t \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(\mathcal{X})_{\mathsf{State}}$) which has the form $\{S_1 \,\&\, \cdots \,\&\, S_n \,\&\, \{IK\}\}$ where & is an associative-commutative union operator with identity symbol $\emptyset$. Each element in the set is either a *strand* S_i or the *intruder knowledge* $\{IK\}$ at that state.

The *intruder knowledge* $\{IK\}$ also belongs to the state and is represented as a set of facts using the comma as an associative-commutative union operator with identity element *empty*. There are two kinds of intruder facts: *positive* knowledge facts (the intruder knows m, i.e., $m{\in}\mathcal{I}$), and *negative* knowledge facts (the intruder *does not yet know* m but *will know it in a future state*, i.e., $m{\notin}\mathcal{I}$), where m is a message expression.

A *strand* [14] specifies the sequence of messages sent and received by a principal executing the protocol and is represented as a sequence of messages $[msg_1^-, msg_2^+, msg_3^-, \ldots, msg_{k-1}^-, msg_k^+]$ such that msg_i^- (also written $-msg_i$) represents an input message, msg_i^+ (also written $+msg_i$) represents an output message, and each msg_i is a term of sort Msg (i.e., $msg_i \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(\mathcal{X})_{\mathsf{Msg}}$).

Strands are used to represent both the actions of honest principals (with a strand specified for each protocol role) and the actions of an intruder (with a strand for each action an intruder is able to perform on messages). In Maude-NPA strands evolve over time; the symbol | is used to divide past and future. That is, given a strand $[\ m_1{}^{\pm},\ \ldots,\ m_i{}^{\pm} \mid m_{i+1}{}^{\pm},\ \ldots,\ m_k{}^{\pm}\ ]$, messages $m_1^{\pm}, \ldots, m_i^{\pm}$ are the *past messages*, and messages $m_{i+1}^{\pm}, \ldots, m_k^{\pm}$ are the *future messages* ($m_{i+1}^{\pm}$ is the immediate future message). A strand $[msg_1^{\pm}, \ldots, msg_k^{\pm}]$ is shorthand for $[nil \mid msg_1^{\pm}, \ldots, msg_k^{\pm}, nil]$. An *initial state* is a state where the bar is at the beginning for all strands in the state, and the intruder knowledge is empty. A *final state* is a state where the bar is at the end for all strands in the state and there is no intruder fact of the form $m{\notin}\mathcal{I}$.

Since the number of states $T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$ is in general infinite, rather than exploring concrete protocol states $[t]_E \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$ Maude-NPA explores *symbolic state patterns* $[t(x_1, \ldots, x_n)]_E \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(\mathcal{X})$ on the free $(\Sigma_{\mathcal{P}}, E_{\mathcal{P}})$-algebra over a set of variables $\mathcal{X}$. In this way, a state pattern $[t(x_1, \ldots, x_n)]_E$ represents not a single concrete state but a possibly infinite set of such states, namely all the *in-*

stances of the pattern $[t(x_1, \ldots, x_n)]_E$ where the variables $x_1, \ldots, x_n$ have been instantiated by concrete ground terms.

The semantics of Maude-NPA is expressed in terms of *rewrite rules* that describe how a protocol moves from one state to another via the intruder's interaction with it. One uses Maude-NPA to find an attack by specifying an insecure state pattern called an *attack pattern.* Maude-NPA attempts to find a path from an initial state to the attack pattern via backwards narrowing (narrowing using the rewrite rules with the orientation reversed). Such a backwards narrowing sequence is called a *backwards path* from to the attack state. Maude-NPA attempts to find paths until it can no longer form any backwards narrowing steps, at which point it terminates. If it at that point it has not found an initial state, the attack pattern is judged *unreachable.* Note that Maude-NPA puts no bound on the number of sessions, so reachability is undecidable in general. Note also that Maude-NPA does not perform any data abstraction such as bound number of nonces. However, the tool makes use of a number of sound and complete state space reduction techniques that help to identify unreachable and redundant states, and thus make termination more likely.

2.3 Never Patterns in Maude-NPA

It is often desirable to exclude certain patterns from transition paths leading to an attack state. For example, one may want to determine whether or not authentication properties have been violated, e.g., whether it is possible for a responder strand to appear without the corresponding initiator strand. For this there is an optional additional field in the attack state containing the *never patterns.* Each never pattern is itself a state pattern. When we provide an attack state A and some never patterns $NP_1, \ldots, NP_k$ to Maude-NPA, every time the tool produces a state S via backwards narrowing from A, it checks whether there is a substitution θ such that $NP_i\theta =_{E_\mathcal{P}} S$. If that is the case, the state is discarded. [2] We will write A with the never patterns $NP_1, \ldots, NP_k$ as $A \mid\mid \text{never}(NP_1) \ldots \mid\mid \text{never}(NP_k)$.

Although never patterns were introduced as a means for specifying authentication properties, they can also be used to reduce the search space. However, we want to preserve completeness as much as possible. Hence we make use of the following results.

Proposition 1. *Let M be a never pattern containing terms of the form $m{\in}\mathcal{I}$. Suppose that the state M is unreachable in Maude-NPA, Then for any state pattern S, if $S \mid\mid never(M)$ is unreachable, then so is S.*

Proof. (Sketch) Suppose that there is a backwards narrowing sequence from S to an initial state. Then it must pass through a state containing $M\theta$ for some substitution θ. But since M is unreachable, so is any state containing $M\theta$.

[2] Maude-NPA also checks whether $NP_i\theta$ satisfies *irreducibility constraints*, as described in [11].

We refer to never patterns that satisfy the conditions of Proposition 1 as *completeness-preserving never patterns.* Given a completeness-preserving never pattern, we can add it to any attack state without affecting its reachability.

In Proposition 2 below, we say that T is a *substate* of S, where T and S are state patterns, if every strand or intruder knowledge statement that appears in T also appears in S.

Proposition 2. *Let* $S_0 \rightsquigarrow_{\sigma_1,R_{\mathcal{P}},E_{\mathcal{P}}^{-1}} S_1 \ldots S_{k-1} \rightsquigarrow_{\sigma_k,R_{\mathcal{P}},E_{\mathcal{P}}^{-1}} S_k = S$ *be a backwards narrowing from an attack pattern* S *to an initial state, and* M *a never pattern containing terms of the form* $m{\in}\mathcal{I}$*, such that for each* S_i *there is no* θ *and* T *such that* $M\theta =_{E_{\mathcal{P}}}= T$*, where* T *is a substate of* S_i*. Then the sequence* $S_0 \rightsquigarrow_{\sigma_1,R_{\mathcal{P}},E_{\mathcal{P}}^{-1}} S_1 \ldots S_{k-1} \rightsquigarrow_{\sigma_k,R_{\mathcal{P}},E_{\mathcal{P}}^{-1}} S_k = S$ *is a backwards narrowing sequence from* $S \mid\mid never(M)$ *to an initial state.*

Proof. (Sketch.) The result follows straightforwardly from the definition of never pattern.

We refer to never patterns that satisfy the conditions of Proposition 2 for a given attack trace as *attack-preserving never patterns.* We can use attack-preserving never patterns to help show that a new version of a protocol is immune to a known attack on the old one. Suppose that we have found an attack on a protocol, and we want to see whether a modified version of the protocol is immune to that attack. Suppose that the search space is intractibly large, even after adding completeness-preserving never patterns. We may be able to reduce the search space by adding attack-preserving never patterns. In that case, unreachability of the attack state with the attack-preserving never patterns does not necessarily imply unreachability of the attack state without these never patterns. But it *does* imply that a specific *class* of attacks, including the original attack we were concerned about, is no longer possible.[3]

We make use of both completeness-preserving and attack-preserving never patterns in our analysis of the IBM-CCA protocols. This is described in more detail in Section 5.

3 IBM CCA API

CCA stands for the Common Cryptographic Architecture API [17] as implemented on the hardware security module IBM 4758, which is an IBM cryptographic coprocessor widely used in security critical systems such as electronic payment and automated teller machine (ATM) networks.

The CCA API contains several protocols, namely the CCA-0 protocol, which is subject to an attack presented by Bond in [3], and other versions of this protocol

[3] Note that if the attack state with the attack-preserving never patterns is reachable, but the original attack is not found, that does not mean that the original attack is not subsumed by any of the found attacks. This is a result of Maude-NPA's state space reduction techniques, which make Maude-NPA produce only some of the possible attacks (but always at least one), when an attack state is reachable.

(CCA-1A, CCA-1B, CCA-2B, CCA-2C, and CCA-2E), designed to avoid this attack.

As explained in [18,9], the CCA is a key management system, which provides commands that use encrypted keys to achieve desired functions. A 168-bit triple-DES key, known as the *master key*, is stored in the security module's tamper-proof memory and is used to encrypt all other keys, which are then kept on the host computer. These other keys, known as *working keys*, are used to perform the various functions provided by the CCA API. There are several types of working keys, depending on the type of action they will be involved in. The CCA API supports the following functions and features:

- Encryption and decryption of data, using the DES algorithm [29].
- Message authentication code (MAC) generation, and data hashing functions.
- Generation and validation of digital signatures.
- Generation, encryption, translation and verification of Personal Identification Number (PIN) and transaction validation messages.
- General key management facilities.
- Administrative services for controlling the initialization and operation of the security module.

The CCA API uses four main types for classifying DES working keys, each of which is further sub-divided into more specific and restrictive types. A working key is stored outside of the security module, encrypted under the exclusive-or of the device's master key and the control vector representing the type of the key. The main key types, and their uses, are as follows:

- **Data Keys**: used for cryptographic operations on arbitrary data.
- **PIN Keys**: used for cryptographic operations on PINs.
- **Key Encryption Keys (KEK)**: used to encrypt and decrypt other working keys during transfer between security modules, and divided into import and export types.
- **Key Generation Keys**: used as input to a key generation algorithm.

The typing mechanism restricts the working keys that can be used for a particular command. For example, the PIN derivation key (PDK) used in the verification of a customer's PIN cannot be used to encrypt arbitrary data.

The following constants and variables are used throughout this section to denote the various control vectors, cryptographic keys and other exchanged data:

- constant `DATA`, `IMP`, `EXP`: control vectors for data, import-type key encryption, and export-type key encryption keys, respectively
- constant `KP`: a part of a key, and not a complete key
- constant `KM`: the security module's master key
- constants `Km1`, `Km2`, and `Km3`: Those are used as a simplification of the CCA protocol where it is assumed that the environment produces the term `e(IMP * KP * KM, Km1 * Km2)`.
- variable *ekek*: an arbitrary key encryption key

- variable eK: a key generation key to encrypt messages
- variable T: an unknown, randomly generated, new cryptographic key or an arbitrary key type control vector. This variable is restricted to constants `DATA`, `IMP`, `EXP` and `PIN`.
- variables *km1*, *km2*, *km3*: i'th key part (used to build an arbitrary key)
- variable X: arbitrary (plain) data

In the following, we provide an informal description of the CCA APIs commands. Table 1 summarizes the exchange of messages performed for each command: messages in the left hand side of the "rule" denote the messages that need to be received; messages in the right hand side denotes messages that are sent as a result of the left hand messages being received. Note that **PKA Symmetric Key Import** is a later addition that converted a public key encryption of **eK** to a symmetric key encryption; it did not appear in the original CCA.

Table 1. CCA API commands and description

API command	**Description**
Encipher	$\mathrm{X}, \{\mathrm{eK}\}_{\{KM*DATA\}} \rightarrow \{\mathrm{X}\}_{eK}$
Decipher	$\{\mathrm{X}\}_{eK}, \{\mathrm{eK}\}_{\{KM*DATA\}} \rightarrow \mathrm{X}$
Key Export	$\{\mathrm{eK}\}_{(KM*T\}}, \mathrm{T}, \{\mathrm{ekek}\}_{\{KM*EXP\}}$ $\rightarrow \{\mathrm{eK}\}_{(ekek*T)}$
Key Import	$\{\mathrm{eK}\}_{(kek*T\}}, \mathrm{T}, \{\mathrm{ekek}\}_{\{KM*IMP\}}$ $\rightarrow \{\mathrm{eK}\}_{\{KM*T\}}$
Key Part Import First	$\mathrm{km1}, \mathrm{T} \rightarrow \{\mathrm{km1}\}_{\{KM*KP*T\}}$
Key Part Import Middle	$\mathrm{km2}, \mathrm{km1}_{\{KM*KP*T\}}, \mathrm{T}$ $\rightarrow (\mathrm{km1} * \mathrm{km2})_{\{KM*KP*T\}}$
Key Part Import Last	$\mathrm{km3}, \mathrm{km2}_{\{KM*KP*T\}}, \mathrm{T}$ $\rightarrow (\mathrm{km2} * \mathrm{km3})_{\{KM*KP*T\}}$
Key Translate	$\{\mathrm{eK}\}_{ekek1*T}, \mathrm{T}, \{\mathrm{ekek1}\}_{KM*IMP},$ $\{\mathrm{ekek2}\}_{KM*EXP} \rightarrow \{\mathrm{eK}\}_{(ekek2*T)}$
PKA Symmetric Key Import	$\{\mathrm{eK}\ ;\ \mathrm{T}\}_{PKA} \rightarrow \{\mathrm{eK}\ \}_{KM*T}$

These commands are explained in more detail below.

- `Encipher` encrypts the given plaintext with the supplied data key. The data key can be either of the general `Key` type, or of one of the subtypes that allow data ciphering.
- `Decipher` decrypts ciphertext which has been encrypted under the supplied data key eK. The data key can be either of the general type, or of one of the subtypes that allow data deciphering.
- `Key Export` converts a working key eK encrypted under the local master key to one encrypted under the supplied export-type key encryption key *ekek*.

- Key Import converts a complete key eK encrypted by the supplied import-type *key encryption key kek* to one encrypted by the local master key KM.
- The Key Part Import commands can be used one after the other, by three different security officers, each in possession of one key part, to create the complete working import key. Note that $km1$, $km2$ and $km3$ are variables.
- Key Translate translates a key eK from encryption by an import key to encryption by an export key.
- PKA Symmetric Key Import converts a complete key eK encrypted by the a public key PKA to one encrypted by the local master key KM.

The exact steps that the security module performs for each command have not been included, since the process is virtually the same in all cases. The master key and all control vectors are known to the security module, and any additional information required is either passed on as a plaintext parameter, or is encrypted under a known key.

Converting these rules to Maude-NPA is straightforward; terms before the arrow are negative terms, and the term after the arrow is positive.

For further details about the specification of the CCA-API commands in Maude-NPA, we refer the reader to [15]. Complete specifications and the analyses outputs may be found in http://www.dsic.upv.es/˜sescobar/Maude-NPA_Protocols/API_Protocols.html

For example, the Maude-NPA version of Key Import is as follows

$$:: nil :: [(e(kek * T, eK))^-, (T))^-, (e(KM * IMP, kek))^-, (e(KM * T, eK))^+]$$

In [3] Bond points out that it is possible to obtain PDK in the clear by combining the commands in an unexpected way. This is described in Table 2, where we have reproduced the attack using the Maude-NPA tool. The terms preceded by a minus sign describe those the attacker needs to know to in order to perform an operation, while the terms preceded by a plus sign describes the term output by an operation.

3.1 IBM's Recommendations to Avoid CCA-0's Attack

In order to prevent the attack of the CCA-0 protocol described above, IBM suggested two recommendations in [16]. In the first one, they recommended the use of a public key version of Key Import, the PKA Symmetric Key Import described above. This version was broken by Cortier et al. in [9]. They then recommended that access control be used, and that no principal be allowed to execute both PKA Symmetric Key Import and Key Import. Following [9] we refer to this as CCA-1. We specify two versions of this in Maude-NPA: CCA-1A in which the attacker has access to Key Import, and CCA-2A, in which the attacker has access to PKA Symmetric Key Import.

In the second recommendation IBM proposed the use of a more elaborate form of *role-based* access control. Principals are assigned to roles determining which commands they are allowed to execute. The goal is to prevent one single

Table 2. Bond's Attack on CCA-0

Exchanged messages	Explanation
+(e(IMP * KP * KM, Km1 * Km2)),	The intruder receives e(IMP * KP * KM, km1 * km2) from the environment.
-(PIN * Km3), -(IMP), -(e(IMP * KP * KM, Km1 * Km2)), +(e(IMP * KM, PIN * Km1 * Km2 * Km3)),	It executes command "Key Part Import Last" where variable km3 is instantiated with Km3 * PIN. In this way he obtains e(IMP * KM, PIN * km1 *km2 * km3).
-(PIN * EXP * Km3), -(IMP), -(e(IMP * KP * KM, Km1 * Km2)), +(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)),	The intruder uses the same command again, this time with variable km3 instantiated with PIN * EXP * Km3, obtaining e(IMP * KM, PIN * EXP * km1 * km2 * km3).
+(e(PIN * Km1 * Km2 * Km3, PDK)),	The intruder receives e(PIN * Km1 * Km2 * Km3, PDK)).
-(e(PIN * Km1 * Km2 * Km3, PDK)), -(null), -(e(IMP * KM, PIN * Km1 * Km2 * Km3)), +(e(KM, PDK)),	When PDK is imported, the intruder uses "Key Import" twice: The first time with inputs e(IMP * KM, PIN * Km1 * Km2 * Km3) and e(PIN * Km1 * Km2 * Km3, PDK) generating the message e(KM, PDK).
-(e(PIN * Km1 * Km2 * Km3, PDK)), -(EXP), -(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)), +(e(EXP * KM, PDK)),	The second time "Key Import" is used with inputs e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3), and e(PIN * Km1 * Km2 * Km3, PDK), which gives the message e(EXP * KM, PDK).
-(e(KM, PDK)), -(null), -(e(EXP * KM, PDK)), +(e(PDK, PDK))	Finally, using the "Key Export" command, the intruder gets e(PDK,PDK).

Table 3. CCA-API commands for IBM's recommendations

API command	CCA-1A	CCA-1B	CCA-2B	CCA-2C	CCA-2E
Encipher	✓	✓	✓	✓	✓
Decipher	✓	✓	✓	✓	✓
Key Export	✓	✓	✓	✓	✓
Key Import	✓		✓		✓
Key Part Import First			✓	✓	
Key Part Import Last				✓	
Key Test		✓	✓		
PKA Sym. Key Import		✓			
Key Translate		✓			

individual from having access to all the commands required to mount Bond's attack. IBM provided an example of the KEK transfer process involving five roles (A-E) such that no single role is able to mount the attack. Following [9], we refer to this as CCA-2. We specify three versions in Maude-NPA depending on which role the attacker is playing; CCA-2B, CCA-2C, and CCA-2E respectively. Since roles A and D do not have access to any of the operations, we do not supply specifications of them.

Table 3 summarizes the commands that each protocol can perform. In the original attack the intruder played the roles C and E together. Note that CCA-XY describes the actions prescribed for Role Y participating in protocol CCA-X.

4 Küsters' and Truderung's XOR-Linear Versions of CCA-Protocols

In [20], Küesters and Truderung analyzed the CCA API protocols in ProVerif via a protocol transformation technique. However, this work does not support full exclusive-or capabilities and requires restricting the analysis to *XOR-linear* protocols; see [20] for details on the XOR-linear property. Because of this, they needed to modify and transform by hand some of the CCA API commands, to produce an XOR-linear protocol equivalent to the original with respect to secrecy properties. We will refer to protocols using these manually-modified commands as "XOR-linear versions" of that protocols.

Table 4. Original specification of the protocol

API command	**Description**
Key Part Import First	km1, T → $\{km1\}_{\{KM*KP*T\}}$
Key Part Import Middle	km2, $km1_{\{KM*KP*T\}}$, T → $(km1 * km2)_{\{KM*KP*T\}}$
Key Part Import Last	km3, $km2_{\{KM*KP*T\}}$, T → $(km2 * km3)_{\{KM*KP*T\}}$
Key Translate	$\{eK\}_{ekek1*T}$, T, $\{ekek1\}_{KM*IMP}$, $\{ekek2\}_{KM*EXP}$ → $\{eK\}_{(ekek2*T)}$

Table 5. Küesters and Truderung version

API command	**Description**
KPI-First + KPI-Add/Middle	km12, T → {KM * KP * IMP}
Key Part Import Last	x, T, KM * KP * T → $(x)_{\{KM*T\}}$ x, IMP → $(X * km12)_{\{KM*IMP\}}$
Key Translate	$\{eK\}_{ekek1*T}$, T, $\{ekek1\}_{KM*IMP}$ → transf(eK,T) transf(eK,T), $\{ekek2\}_{\{KM*EXP\}}$ → $\{eK\}_{(ekek2*T)}$

As we can see in Tables 4 and 5, the XOR-linear versions of the CCA operators are as follows. The "`KPI-First + KPI-Add/Middle`" and "`Key Part Import Last`" API commands are the XOR-linear equivalent to the original "`Key Part Import First`", "`Key Part Import Middle`" and "`Key Part Import Last`" commands. Note that now the "`Key Translate`" command requires two steps instead of one in the original version. All the other commands remain the same without any transformation.

5 Maude-NPA's CCA Analysis

In this section we describe the results of Maude-NPA's analysis of both the original CCA protocols as proposed by IBM and the XOR-linear versions of Küesters and Truderung. We did not analyze the versions for ProVerif that were produced automatically from the XOR-linear versions, since these did not use XOR nor AC. In each case we asked if the attacker could learned $e(PDK, PDK)$, since this is the information that Bond's attacker needs to compute PDK. In the following we give the results of our Maude-NPA analyses, comparing the results for the original protocol with those for its XOR-linear transformation when one exists. The results are given in Table 7, where the number of states at depth N is the number of different N-length backwards narrowing sequences produced after N backwards narrowing steps. Complete specifications and the analyses outputs may be found in `http://www.dsic.upv.es/ sescobar/Maude-NPA_Protocols/API_Protocols.html`.

In some cases, we were not able to obtain termination unaided, and were required to use never patterns as follows. We use two completeness-preserving never patterns: (i) `e(Key, KM * Msg) inI` and (ii) `e(IMP * KM, Type * Key) inI`. These have been proved unreachable in Maude-NPA. We also use several attack-preserving never patterns. One of these is (iii) `PDK inI`. This is motivated by the fact that in Bond's attack the intruder is trying to find $e(PDK, PDK)$ so that it can learn PDK, so we would not expect it to have learned PDK already. The others are different terms of the form $(X * Y) \in \mathcal{I}$ not used in Bond's attack. These are: (iv) $(Km1 * Y) \in \mathcal{I}$, (v) $(Km2 * Y) \in \mathcal{I}$, (vi) $(PDK * Y) \in \mathcal{I}$, (vii) $(KM * Y) \in \mathcal{I}$, and (viii) $(Y * e(K, Y)) \in \mathcal{I}$ where `K` and `Y` are variables. In Table 7 we give the cases in which we use and do not use never patterns. Since the protocols are similar, we use the same never patterns for all cases.

CCA-0. The CCA-0 protocol is insecure, since it is subject to the attack found by Bond in [3]. In this attack the intruder obtains a PIN derivation key in the clear, as in the IBM attack and, thus, can compute PINs from bank account numbers. This attack is the same found by Küesters and Truderung in [20].

Using the same assumptions as in [3] in terms of the role played by the intruder and its knowledge, Maude-NPA finds the attack of the CCA-0 protocol after 7 steps of protocol analysis. Table 6 shows the numbers of states generated at each depth of the backwards reachability analysis from an attack state in which the intruder has learned the expression `e(PDK,PDK)`.

As we can see from rows 1 and 2 of Table 7, Maude-NPA finds the initial state for both protocols, the original CCA-0 and XOR-linear version, at the same depth of the backwards search tree. If the analysis is continued, the XOR-linear version produces a finite search space containing 2495 states in total. The use of the never patterns was required to guarantee termination for the more complex original protocol, but not the XOR-linear version. For both protocols Maude-NPA terminated at Step 7; that is, it produced no new states at Step 8.

CCA-1A. This protocol is XOR-linear and Küesters and Truderung do not transform it. Row 3 of Table 7 summarizes the result of the analysis of the pro-

Table 6. CCA-0 Analysis Output

Level	1	2	3	4	5	6	7
States	1	7	27	79	89	44	1
Solutions	0	0	0	0	0	0	1

Table 7. Experimental results

	Protocol	**States**	**Depth**	**Terminates**
1	CCA-0	291^{*}	7	Yes
2	CCA-0-XOR-linear	2495	7	Yes
3	CCA-1A	21^{*}	5	Yes
4	CCA-1B	48^{*}	6	Yes
5	CCA-1B-XOR-linear	1	2	Yes
6	CCA-2B	324^{*}	11	Yes
7	CCA-2C	131^{*}	6	No
8	CCA-2C-XOR-linear	105	4	No
9	CCA-2E	385^{*}	7	No

***This protocol analysis uses never patterns**

tocol specified in Maude-NPA using never patterns. The search space terminates at step 5 (that is there are no states produced at step 6).

CCA-1B. This protocol is not XOR-linear and Küesters and Truderung manually transformed it. In Table 8 we can see the differences between the two CCA-1B protocols, the original and the XOR-linear versions. Rows 4 and 5 of Table 7 summarize the results of the analysis of both versions. As we can see, the XOR-linear version is extremely simple and the analysis is almost immediate in Maude-NPA, requiring no never patterns. The search space terminates at depth 6, finding no initial state.

CCA-2B. Row 6 of Table 7 summarizes the result of the analysis of CCA-2B. Note that this protocol is XOR-linear and Küesters and Truderung do not transform it. The search, using never patterns, terminates at depth 11, finding no initial state.

CCA-2C. Table 8 shows the original protocol and the XOR-linear version provided by Küesters and Truderung in [20]. Rows 7 and 8 of Table 7 summarize the results of the analysis of both versions. In these two cases we were not able to run Maude-NPA to termination. For the XOR-linear version we were able to run Maude-NPA to depth 4, and depth 6 in the original version.

CCA-2E. Row 9 of Table 7 summarizes the result of the analysis of the protocol specified in Maude-NPA. Note that this protocol is XOR-linear and Küesters and Truderung do not transform it. In this case we were able to run Maude-NPA to depth 7, but were not able to achieve termination.

In all the cases in which we were unable to achieve termination, the issue does not seem to be so much state explosion as the time to produce all the states at a given depth. Indeed, in the case of CCA-2E, Maude-NPA found 76 states at step 6 and 54 states at step 7, suggesting that it might have terminated if run to a greater depth.

Table 8. Original and Küesters-Truderung versions of CCA-1B and CCA-2C

API command	**Description**
CCA 1-B Original Key Translate	{eK}$_{ekek1*T}$, T, {ekek1}$_{KM*IMP}$, {ekek2}$_{KM*EXP}$ → {eK}$_{(ekek2*T)}$
CCA-1B-XOR-linear Key Translate	{eK}$_{ekek1*T}$, T, {ekek1}$_{KM*IMP}$ → transf(eK,T) transf(eK,T), {ekek2}$_{\{KM*EXP\}}$ → {eK}$_{(ekek2*T)}$
CCA-2C Original Key Part Import Last	km3, (km2)$_{\{KM*KP*T\}}$, T → (km2 * km3)$_{\{KM*KP*T\}}$
CCA-2C-XOR-linear Key Part Import Last	x, T, KM * KP * T → (x)$_{\{KM*T\}}$ x, IMP → (X * km12)$_{\{KM*IMP\}}$

6 Discussion

We have demonstrated that in certain cases Maude-NPA is indeed able to prove properties of XOR-based cryptographic APIs. This is to the best of our knowledge the first application of a general-purpose unbounded session cryptographic protocol analysis tool that directly models the properties of XOR to XOR-based cryptographic APIs. However, there were a number of performance issues that affected termination. We discuss these in more detail below, and what can be done to address them.

We do not provide a detailed comparison of the performance of the different tools, since the way protocols are modeled and security properties proved vary from case to case. For example, Cortier et al. analyze a slightly different version of CCA-2 in which principals are given greater privileges. Also, each of the analyses of Cortier et al., Küsters and Truderung, and ourselves makes different assumptions about the initial knowledge available to the attacker. On the other hand, we can make some general comparisons. Küsters and Truderung are able to achieve termination in all cases, although this comes at a cost, since it is not clear that all protocols can be converted to XOR-linear versions, and it is unknown whether the conversion process can be made sound and complete with respect to authentication as well as secrecy properties. Cortier et. al. are able to obtain termination for CCA-1, but for CCA-2 they ran their algorithm only up to a certain bound, and then verified informally that the attacker could not gain any useful terms by interacting further with the protocol.[4] Maude-NPA terminated for CCA-0, CCA-1, and CCA-2B with never patterns, and for the XOR-linear versions without. However, it had more problems with CCA-2C and CCA-2E, even the XOR-linear version. We note that state explosion could be controlled with never patterns; the main problem we were experiencing was that Maude-NPA took longer and longer to complete its search at a given depth, even though it might not be producing that many states.

The chief cause of state explosion seems to be the failure of Maude-NPA state space reduction techniques, in particular the Maude-NPA *grammars* to deal with complex combinations of exclusive-or expressions. Maude-NPA uses inductive techniques to recognize terms that are unlearnable by the intruder, and generates grammars that describe these terms. It works well with most theories but occasionally has problems with XOR and Abelian group theories, especially when they occur many times in a protocol, as they do in IBM CCA. We are currently reassessing our grammar generation techniques in the light of our experience with IBM CCA.

The fact that Maude-NPA is taking a long time to complete, even when it does not produce that many states, means that it is generating many states which are subsequently rejected as unreachable or redundant using the state space reduction techniques. Paradoxically, this behavior could be improved by improving the state space reduction techniques, since if unreachable states are removed earlier, less states are generated later on. Performance may also be improved by sound and complete transformations to simpler protocols. For example, Küster's and Truderung's transformations to XOR-linear protocols generally resulted in protocols that were easier for Maude-NPA to analyze, and although they were done manually, they use a general strategy that could possibly be automated. We could perhaps employ similar techniques to produce protocols that are "small" with respect to an XOR-complexity metric, rather than XOR-linear.

Finally, we note the contribution made to this work by never patterns. As far as we know, this is the first work that considers their effect on soundness and completeness. Completeness-preserving never patterns have the potential to be a valuable tool for use as an additional state space reduction technique. Indeed it should be fairly straightforward to prove that a pattern unreachable and add it automatically to a specification as a completeness-preserving never pattern; this was indeed a feature of the NPA tool [24] that preceded Maude-NPA. A more ambitious plan would be to search automatically for completeness-preserving never patterns, e.g. by finding patterns that keep on occurring in unreachable states, testing them for unreachability, and adding them as never patterns if they past the test.

[4] It is not clear from [9], whether performance was the chief factor in not choosing a higher bound.

7 Conclusions

We have specified, analyzed and compared different versions of the IBM CCA API protocols. This is to the best of our knowledge the first application of a general-purpose unbounded session cryptographic protocol analysis tool that directly models the properties of XOR to XOR-based cryptographic APIs. We have identified the bottlenecks and performance issues and have outlined plans for handling them. Finally, we have introduced the notion of *completeness-* and *attack-preserving* never patterns as a new means of controlling the size of the search space, and have outlined plans for automating their use.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. ACM Trans. Inf. Syst. Secur. 10(3) (2007)
2. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW 2014), Cape Breton, Nova Scotia, Canada, June 2001, pp. 82–96. IEEE Computer Society (2014)
3. Bond, M.: Attacks on cryptoprocessor transaction sets. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 220–234. Springer, Heidelberg (2001)
4. Butler, F., Cervesato, I., Jaggard, A.D., Scedrov, A.: A formal analysis of some properties of kerberos 5 using msr. In: CSFW, pp. 175–1790. IEEE Computer Society (2002)
5. Cachin, C., Chandran, N.: A secure cryptographic token interface. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, pp. 141–153 (2009)
6. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: An NP decision procedure for protocol insecurity with XOR. In: 18th Annual IEEE Symposium on Logic in Computer Science, LICS 2003 (2003)
7. Comon-Lundh, H., Shmatikov, V.: Intruder deductions, constraint solving and insecurity decision in presence of exclusive-or. In: 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003), pp. 271–280 (2003)
8. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 148–164. Springer, Heidelberg (2003)
9. Cortier, V., Keighren, G., Steel, G.: Automatic analysis of the aecurity of XOR-based key management schemes. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 538–552. Springer, Heidelberg (2007)
10. Cortier, V., Steel, G.: A generic security API for symmetric key management on cryptographic devices. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 605–620. Springer, Heidelberg (2009)
11. Erbatur, S., et al.: Effective Symbolic Protocol Analysis via Equational Irreducibility Conditions. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 73–90. Springer, Heidelberg (2012)

12. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2007)
13. Escobar, S., Meadows, C., Meseguer, J., Santiago, S.: Sequential Protocol Composition in Maude-NPA. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 303–318. Springer, Heidelberg (2010)
14. Thayer Fabrega, F.J., Herzog, J., Guttman, J.: Strand Spaces: What Makes a Security Protocol Correct? Journal of Computer Security 7, 191–230 (1999)
15. González-Burgueño, A.: Protocol Analysis Modulo Exclusive-Or Theories: A Case study in Maude-NPA. Master's thesis, Universitat Politècnica de València (March 2014), `https://angonbur.webs.upv.es/Previous_work/Master_Thesis.pdf`
16. IBM. Comment on Mike's Bond paper A Chosen Key Difference Attack on Control Vectors (2001), `http://www.cl.cam.ac.uk/~mkb23/research/CVDif-Response.pdf`
17. IBM. CCA basic services reference and guide: CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 (2001), `http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf.2008`
18. Keighren, G.: Model Checking IBM's Common Cryptographic Architecture API. Technical Report 862, University of Edinburgh (October 2006)
19. Kemmerer, R.A.: Using formal verification techniques to analyze encryption protocols. In: IEEE Symposium on Security and Privacy, pp. 134–139. IEEE Computer Society (1987)
20. Küsters, R., Truderung, T.: Reducing protocol analysis with xor to the xor-free case in the horn theory based approach. J. Autom. Reasoning 46(3-4), 325–352 (2011)
21. Linn, J.: Generic security service application program interface version 2, update 1. IETF RFC 2743 (2000), `https://datatracker.ietf.org/doc/rfc2743`
22. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers & Security 11(1), 75–89 (1992)
23. Meadows, C.: Applying formal methods to the analysis of a key management protocol. Journal of Computer Security 1(1) (1992)
24. Meadows, C.: The NRL protocol analyzer: An overview. Journal of Logic Programming 26(2), 113–131 (1996)
25. Meadows, C., Cervesato, I., Syverson, P.: Specification and Analysis of the Group Domain of Interpretation Protocol using NPATRL and the NRL Protocol Analyzer. Journal of Computer Security 12(6), 893–932 (2004)
26. Meadows, C.: Analysis of the internet key exchange protocol using the nrl protocol analyzer. In: IEEE Symposium on Security and Privacy, pp. 216–231. IEEE Computer Society (1999)
27. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic snalysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013)
28. Mukhamedov, A., Gordon, A.D., Ryan, M.: Towards a verified reference implementation of a trusted platform module. In: Christianson, B., Malcolm, J.A., Matyáš, V., Roe, M. (eds.) Security Protocols 2009. LNCS, vol. 7028, pp. 69–81. Springer, Heidelberg (2013)
29. National Institute of Standards and Technology. FIPS PUB 46-3: Data Encryption Standard (DES), supersedes FIPS 46-2 (October 1999)

30. Nieuwenhuis, R. (ed.): CADE 2005. LNCS (LNAI), vol. 3632. Springer, Heidelberg (2005)
31. Steel, G.: Deduction with xor constraints in security api modelling. In: Nieuwenhuis (ed.) [30], pp. 322–336
32. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational horn clauses. In: Nieuwenhuis (ed.) [30], pp. 337–352