Detecting Laser Fault Injection for Smart Cards Using Security Automata

Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team – University of Limoges
123 Avenue Albert Thomas, 87060 Limoges CEDEX, France
`{guillaume.bouffard,bhagyalekshmy.narayanan-thampi}@xlim.fr,`
`jean-louis.lanet@unilim.fr`

Abstract. Security and interoperability issues are increasing in smart card domain and it is important to analyze these issues carefully and implement appropriate countermeasures to mitigate them. Security issues involve attacks on smart cards which can lead to their abnormal behavior. Fault attacks are the most important among them and they can affect the program execution, smart card memory, etc. Detecting these abnormalities requires some redundancies, either by another code execution or by an equivalent representation. In this paper, we propose an automatic method to provide this redundancy using a security automaton as the main detection mechanism. This can enforce some trace properties on a smart card application, by using the combination of a static analysis and a dynamic monitoring. The security officer specifies the fragments of the code that must be protected against fault attacks and a program transformer produces an equivalent program that mesh a security automaton into the code according to the security requirements.

Keywords: Fault attacks, Trust, Smart Card, Security Automata, Countermeasure.

1 Introduction

Smart card is a small embedded chip/device which is commonly used in our day to day life for serving various purposes in banking, electronic passports, health insurance card, pay TV, SIM card, etc. It has efficient computing capabilities and security features for ensuring secure data transaction and storage. Many hardware and software attacks are performed to gain access to the assets stored inside a smart card. Since it contains sensitive information, it must be protected against attacks. The laser faults are the most difficult among them to be handled.

Fault Injection (FI) attacks can cause the perturbation of the chip registers (e.g., the program counter, the stack pointer, etc.), or the writable memory (variables and code modifications). If these perturbations are not detected in advance, an attacker can get illegal access to the data or services. Some redundancy is necessary to recognize the deviant behavior which can be provided by a security automaton and a reference monitor. This technique has emerged as a powerful and flexible method for enforcing security policies over untrusted code. The process verifies the dynamic security checks or a call to the security functions into the untrusted code by monitoring the evolution of a state machine.

In our work, we propose to implement the transition functions of such a state machine natively in the Java Card Virtual Machine (JCVM). For interoperability reasons, we implement a less efficient API to replace the transition functions and a static analyzer to verify the coherence of the security property.

This paper is organized as follows: section two describes the FI attacks on smart cards and their effects on program execution. The known detection mechanisms and their comparison are discussed in the third section. Section four presents our contributions and countermeasures. Final section gives the conclusions of our work.

2 Related Works

Aktug [1] defined a formal language for security policy specifications *ConSpec*, to prove statically that a monitor can be inlined into the program byte code, by adding first-order logic annotations. A weakest precondition computation was used here which works as same as the annotation propagation algorithm employed in [14] to produce a fully annotated, verifiable program for the Java Card. This allows the use of JML verification tools, to check the actual policy adherence. Such a static approach cannot be adopted here due to the dynamic nature of the attack.

As far as we know, the only application of the security automaton for smart card was presented in [13] where the concept of policy automaton which combines defeasible logic with the state machine. It represents complex policies as combinations of basic policies. A tool has been implemented for performing policy automaton analysis and checking policy conflicts and a code generator is used to implement the transition functions that creates a Java Card applet. It was concerned mainly to enforce invariants in the application.

3 Faults on Smart Cards

In general, a fault is an event that changes the behavior of a system such that the system no longer provides the expected service. It may not be only an internal event in the system but also, a change in the environment that causes a bit flip in the memory. However the fault (when activated), is the primary reason for the changes in the system that leads to an error which in turn causes a failure of the complete system. In order to avoid such a failure, faults have to be detected as early as possible and some actions must be carried out to correct or stop the service. Thus, it is necessary to analyze the errors generated by these faults more precisely. In the current smart card domain, fault attacks are the most difficult attacks to be tackled.

3.1 Fault Attacks

Smart card is a portable device which requires a smart card reader (provides external power and clock sources) to operate it. The reader can be replaced with specific equipment to perform the attacks. With short variations of the power supply it is possible to induce errors into the internal operations of the smart card. These perturbations are called spike attacks, which may induce errors in the program execution. Latter aims at confusing the program counter which can cause the improper working of conditional checks, decrease in loop counters and the execution of the arbitrary instructions. The reader like MP300 can also be used to provide glitch attack. A glitch [3,7,12] incorporates short deviations beyond the required tolerance from a standard signal bounds. They can be defined by a range of different parameters and can be used to inject memory faults as well as faulty execution behavior. Hence, the possible effects are same as in the spike attacks. If the chip is unpacked, such that the silicon layer is visible, it is possible to use a laser to induce perturbation in the memory cells [8]. These memory cells have been found to be sensitive to light. Due to photoelectric effect, modern lasers can be focused on relatively small regions of a chip, so that FI can be targeted well [4].

To prevent the occurrence of FI attacks, it is necessary to know its effects on the smart card. FI models have been already discussed in details in [6,20]. A widely accepted model corresponds to an attack that changes one byte at a precised and synchronized time [19]. An attack using the precise bit error model had been described in [18] which is not realistic on current smart cards due to the implementation of hardware security on memory (error correction and detection code or memory encryption) of modern components.

In real time, an attacker physically injects energy into a memory cell to switch its state. Thus up to the underlying technology, the memory will physically takes the value `0x00` or `0xFF`. If memory is encrypted, the physical value becomes a random value[1].

3.2 Effects of Fault Attacks on the Program Execution

In this work, only a single fault will be considered. The proposed mechanism supports dual faults if the automaton is protected by some checksum method. An attacker can break the confidentiality and/or the integrity mechanisms incorporated in the card. The code integrity of the program ensures that the original installed code is the same as that executed by the card. The attacker can modify the value returned by a function to allow the execution of sensitive code without authorization. He can also generate a faulty condition to jump to a specific statement, avoid a given method invocation or ignore a condition loop.

The data of a program are also a sensitive asset to be protected. With a single fault, an attacker can permanently or temporarily modify sensitive information.

[1] More precisely, a value which depends on the data, the address, and an encryption key.

In particular, it can affect the variables used in any evaluation instruction like never start a loop, ignore initialization and so on. The smart card should ensure the confidentiality of the assets. The attacker may modify the data to be copied, from the application byte array or to the I/O smart card buffer by modifying the address of the buffer. Another way to obtain the asset is to change the number of bytes to be send in the buffer. This overflow provides information of data that follow the bytes sent from the application.

3.3 Fault Detection Mechanisms

The mechanisms for fault injection detection can be classified in to three countermeasure approaches: static, dynamic and mixed.

Static Countermeasure Approach. Static countermeasures ensure that each test is done correctly and/or the program Control Flow Graph (CFG) remains unchanged as described by the developer.

To verify if a test i.e., (a sensitive condition `if`) is done correctly, the usage of a redundancy *if-then-else* statement should improve the statement branching security. Indeed, if a fault is injected during an `if` condition, an attacker can execute a specific statement without a check. In real time, a second-order FI is difficult with a short delay between two injections. A second-order `if` statement is used to verify the requirements needed to access a critical operation to prevent the faulty execution of an *if-then-else* statement. An example of this kind of implementation is listed in the Listing 1.1 Second-order if statement.

Listing 1.1. Second-order if statement

```
// condition is a boolean
if(condition) {
  if(condition) {
    // Critical operation
  } else {/*Attack detected!*/
      }
} else {
  if(!condition) {
    // Access not allowed
  }else{/*Attack detected!*/}}
```

Listing 1.2. Step counter

```
short step_counter=0;
if(step_counter==0) {
  // Critical operation 1
  step_counter++;
} else {/*Attack detected!*/
    }
/* ... */
if(step_counter==1) {
  // Critical operation 2
  step_counter++;
}else{/*Attack detected!*/}
```

The problem with a second-order `if` condition is that the program CFG is not guaranteed. To ensure it, the developer can implement a step counter as described in the Listing 1.2. With this method, each node of the CFG, defined by the developer is verified during the runtime. If a node is executed with a step counter set with a wrong value, an incorrect behavior is detected.

Dynamic Countermeasure Approach. The smart card can implement countermeasures on dynamic elements (stack, heap, etc.) and thereby ensure integrity to prevent the modification of them. A checksum can be used to verify the manipulated value for each operation. Another low cost countermeasure approach, to protect stack element against FI attack was explained by Dubreuil *et al.* in [9]. Their countermeasure implements the principle of a dual stack where each value is pushed from the bottom and growing up into the element stack. In contrary, each reference is pushed from the top and growing down. This countermeasure protects smart card against type confusion attack.

As described before, a program's code is also an asset to be protected. In order to ensure the code confidentiality, the memory may be encrypted. For using a more affordable countermeasure, Barbu explained [4], a solution where the code is scrambled. Unfortunately, a brute force attack can bypass a scrambled memory. Razafindralambo *et al.* proposed in [16] a randomized scrambling which improves the code confidentiality.

Enabling all countermeasures during the whole program execution is not necessary and also it is more costlier for the card. Hence, to reduce the implementation cost of the countermeasure, Barbu *et al.* [5] described user-enabled countermeasure(s) where the developer can decide to enable a specific countermeasure for a code fragment.

Recently, Farissi *et al,* presented [10] an approach based on artificial intelligence and in particular neural networks. This mechanism is included in the JCVM. After a learning step, this mechanism can dynamically detect abnormal behavior of each smart card's program.

Mixed Countermeasure Approach. Unlike previous approaches, mixed methods use off-card operations where some computations are done for embedded runtime checks. This way offers a low cost because costly operations are realized outside the card.

To ensure the code integrity, Prevost *et al.* patented [15] a method where for each basic blocks of a program, a hash value is computed. The program is sent to the card with each basic block's hash. During the execution, the smart card verifies this value for each executed basic block and if a hashsum is wrong, an abnormal behavior is detected.

Sere [2], described three countermeasures, based on bit field, basic block and path check, to protect smart card against FI attacks. These countermeasures require off-card operations done during the compilation step to compute enough information to be provided to the smart card through a custom component. The smart card dynamically checks the correctness of the current CFG. Since there are off-card operations, this countermeasure has a low footprint in the smart card's runtime environment.

4 Security Automata and Execution Monitor

Detecting a deviant behavior is considered as a safety property, *i.e.* properties that state *nothing bad happens.* A safety property can be characterized by a set of disallowed finite execution based on regular expression. The authorized execution flow is a particular safety property which means that, the static control flow must match exactly the runtime execution flow without attacks. For preventing such attacks, we define several partial traces of events as the only authorized behaviors. A key point is that this property can be encoded by a finite state automaton, while the language recognized will be the set of all authorized partial traces of events.

4.1 Principle

In [17], Schneider defined a security automaton, based on Büchi automaton as a triple (Q, q_0, δ) where Q is a set of states, q_0 is the initial state and δ a transition function δ: (Q $\times$ I) $\rightarrow 2^Q$. The set S is the input symbols, *i.e.* the set of security relevant actions. The security automaton processes a sequence of input symbols s_1, s_2, $\ldots$ and the sequence of symbols is read as one input at a time. For each action, the state is evaluated by starting from the initial state s_0. As each s_i is read, the security automaton changes Q' in $\cup_{q \in Q'} \delta(s_i, q)$. If the security automaton can perform a transition according to the action, then the program is allowed to perform that action, otherwise the program is terminated. Such a mechanism can enforce a safety property as in the case for checking the correctness of the execution flow.

The property we want to implement is a redundancy of the control flow. In the first approach, the automaton that verifies the control flow could be inferred using an interprocedural CFG analysis. In a such a way, the initial state q_0 is represented by any method's entry point. S is made of all the byte codes that generate a modification of the control flow along with an abstract instruction *join* representing any other instructions pointed by a label. By definition, a basic block ends with a control flow instruction and starts either by a first instruction after control flow instructions or by an instruction preceding a label. When interpreting a byte code, the state machine checks if the transition generates an authorized partial trace. If not, it takes an appropriate countermeasure.

The transition functions are executed during byte code interpretation which follows the isolation principle of Schneider. Using a JCVM, it becomes obvious that the control of the security automaton will remain under the control of the runtime and the program cannot interfere with automaton's transitions. Thus, there is no possibility for an attacker to corrupt the automaton because of the Java sandbox model. Of course, the attacker can corrupt the automaton using the same means as he corrupted the execution flow. By hypothesis, we do not consider actually the double FI possibility for the attacker. If needed, it is possible to protect the automaton with an integrity check verified before each access to the automaton.

4.2 Implementation in a Java Card Virtual Machine

The control of the transition functions is quite obvious. Once the automaton array has been built during the linking process, each Java frame is improved with the value of the current state q_i. In the case of a multithreaded virtual machine, each thread manages the state of the current method security automaton in its own Java frame for each method.

Listing 1.3. Transition function for the `IFLE` byte code (next instruction)

```
 1 int16 BC_ifle(void) {
 2     if (SM[frame->currentState][INS] != *vm_pc)
 3         return ACTION_BLOCK;
 4     vm_sp--;
 5     if (vm_sp[0].i <= 0) return BC_goto();
 6     if (SM[frame->currentState][NEXT] != state(vm_pc))
 7         return ACTION_BLOCK;
 8     vm_pc += 2;
 9     frame->currentState = SM[frame->currentState][NEXT];
10     return ACTION_NONE; }
```

The automaton is stored as an array with several columns like the next state, the destination state and the instruction that generates the end of the basic blocks. In the Listing 1.3, the test (in line 2) verifies that the currently executed byte code is the one stored in the method area. According to the fault model, a transient fault should have been generated during the instruction decoding phase. If it does not match, the JCVM stops the execution (line 3). If the evaluation condition is true, it jumps to the destination (line 5). Else, it checks if the next Java program pointer is a valid state for the current state of the automaton. If it is allowed, the automaton changes its state.

Listing 1.4. Transition function for the `IFLE` byte code (target jump)

```
1 int16 BC_goto(void) {
2     vm_pc = vm_pc - 1 + GET_PC16;
3     if (SM[frame->currentState][DEST] != state(vm_pc))
4         return ACTION_BLOCK;
5     frame->currentState = SM[frame->currentState][DEST];
6     return ACTION_NONE; }
```

In Listing 1.4, the last part of the `IFLE` byte code checks also if the destination Java program counter matches with the next state and update the current state.

Listing 1.5. Decoding an instruction

```
1     while (true) { handler = bytecode_table[*vm_pc];
2                    vm_pc++; bc_action = handler();
```

In the decode phase of the instruction, the laser can hit the data bus while transferring the needed information from the memory. In this JCVM decode phase (Listing 1.5), the address of the byte code function is obtained in line 1. At that time, either the `vm_pc` or the `handler` can be corrupted. Thus, the byte code being executed is not the one stored in the memory. Therefore, we need to check the execution instruction is the same as that of the stored one.

The security automaton is build during the linking process of the Java Card applet. During the linking step, the method is processed byte code by byte code linearly, allowing to build the automaton array. Each state s_i corresponds to `vm_pc` start and `vm_pc` end; the function `state(vm_pc)` returns an index of the array corresponding to the state that includes the `vm_pc`.

Here, we presented the basic security automaton of the control flow redundancy which needs to be improved. In section 3.3, we have seen the possibility to skip an instruction or a call to a function. The granularity of the basic block is not enough to handle this issue and checking each instruction is not realistic. So, we need to find a trade off between the granularity and each instruction. We propose to insert calls to an abstract function `setState()`, in some sensitive code fragments. Being a control flow function, a call to a function will be directly included into the security automaton. Even if the latter is empty a call to a function costs a lot due to the built and the destruction of the frame. For that purpose, we developed a byte code analyzer that emulates this call. It takes the input as binary file (the Cap file) and extract all occurrences of the `invokestatic` instruction and replaces them by a simple `goto` instruction to the next line. It has the same semantics, *i.e.* an entry in the automaton array but it costs much less.

Listing 1.6. Inserting checks in a Java Card basic block

```
 1 apdu.setIncomingAndReceive();
 2 Util.arrayCopy(apduBuffer, (short) (ISO7816.OFFSET_CDATA), D,
 3                   (short) 0, (short) 4);
 4 setState();
 5 if (b == false) ISOException.throwIt(error_date);
 6 tempo = Util.getShort(A, (short) 0);
 7 setState();
 8 k = 1;
 9 (short) ((tempo >> k) / NbAmounts[0])
10 setState();
11 functionR(A, key, D);
12 setState();
```

The code presented in Listing 1.6 is extracted from the Internet protocol payment defined by Gemalto in [11]. We ensure that, the JCVM verifies if each step has been correctly passed in line 4, 7, 10 and 12. This corresponds to what is usually done in a Java Card secured development with step counters as shown in the Listing 1.2. However, it is integrated in a more general framework to automate the fault detection.

The Lightweight Version in the INOSSEM Project. The aim of the project INOSSEM is to guarantee a security interoperability between several smart card manufacturers. The security specification must be independent to the design of the JCVM. For that purpose, it has been decided to design the countermeasures as a Java API defining all the required services. The security automaton is one of the INOSSEM classes. The main drawback of this approach versus our native implementation is the cost of all the function calls. But on the other hand, the code fragment protected with the security automaton can be isolated from the rest of the application.

The call to the API methods that manages the transition functions of the security automaton should be explicitly written by the developer. The developer should only insert the call to the method `setState()` in his applet to have the guarantee that the JCVM will verify the control flow of this code fragment. A call to `endStateMachine()` checks the correct ending of the security automaton. Being under the control of the Java Card runtime, the isolation principle is still respected during the execution of the API.

The main difference is that the construction of the security automaton is delegated to the developer. He has to examine all the authorized traces and build the automaton. Exceptions that should occur during execution must also be a part of the authorized traces. The developers know which parts of their application is sensible and they focus on the protection of a particular code fragment. This leads to a new issue, the coherence of the security policy defined by the automaton.

Coherence of the Policy. If a developer specifies a security automaton that partially represents the CFG, the automaton could consider some illegal transitions while they are legal traces. In the example given in Fig. 1, we have the CFG at the left side and the specification of the security automaton at the right side.

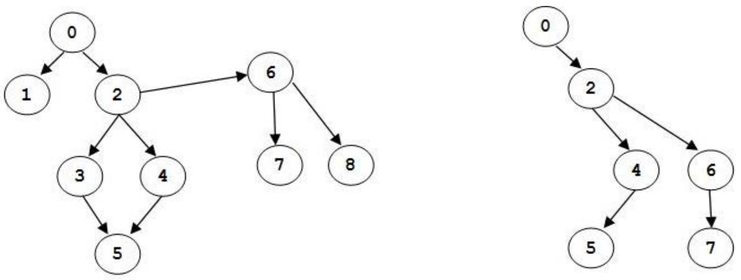

Fig. 1. Partial specification

The developer is only interested to protect some execution paths. He defines the following state sequences $\{q_0, q_2, q_4, q_5\}$ and $\{q_0, q_2, q_6, q_7\}$. If the sequence $\{q_0, q_2, q_3, q_5\}$ is executed, then in the state q_2, the automaton has to process a transition to the state q_3. It terminates the execution of the target while the execution path is valid. In fact, the security policy must be a subgraph of the control flow and thus, an edge must be added to the security automaton between the state q_2 and q_5. The verification of the coherence algorithm must check that the security automaton is a subgraph of the CFG. If the security automaton is a partial subgraph (i.e. every edges of the state of a CFG are not included), then the missing edges must be added to it. A specific action is to be done while

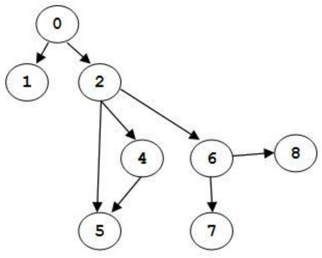

Fig. 2. Complete specification

reaching the end of the security automaton. Thus, all the terminal states must be added to the security automaton (q_1, q_8). The correct security automaton is given in Fig 2.

Each basic block can be split into several states as shown in Listing 1.6. For example, the state q_4 can be made of the sequence $\{q_4', q_4'', q_4''', \ldots\}$. In such a case, the security automaton is no more a subgraph of the CFG. While adding the edge for the closure, this sequence must be recognized as the state q_4. It is obvious to control the coherence for a simple automaton, like the one presented here. For real life examples, we need a static analyzer to check the coherence and build the automaton.

4.3 Static Analyzer and Code Meshing

Of course, this process is manageable automatically by a static analyzer before loading it into the card. The analyzer (SA) extracts the CFG (inter-class and interprocedural analysis) of the program from the source code, defining all the basic blocks. Then, it extracts the security automaton (call the API methods `setState()` and `endStateAutomaton()`) recognizing the state extension by subsequences when these calls are included within a basic block. It checks by comparing the security automaton and CFG if closure edges are missing and proposes through a graphical interface to the user to add the missing edges. The second step consists of initializing the state automaton object, defining all the states as final static fields and calling all the missing methods of the API: the `endStateMachine()` at the end of all the sink states.

Our prototype proposes also the possibility for the cards that do not implement the INOSSEM API to inline the security automaton into the application code with the two transition functions as shown in the Listing 1.7. Due to the fact that Java Card does not support multidimensional arrays, the security automaton must manage the index to simulate a matrix. The SA-Analyzer has filled the `securityAutomaton` array either with an index or the value `NO_STATE`. Then, if a `NO_STATE` is found for a transition, the method throws an exception.

The complexity of `setState()` is $\theta(1)$ while the complexity of `endStateMachine()` is $\theta(n)$, n being the maximum neighbors for all the nodes of the graph.

Listing 1.7. Inserting `setState` method

```
public void setState(short state) {
    if (securityAutomaton[(currentState*NB_COL)+state] !=
        NO_STATE)
        currentState = securityAutomaton
                            [currentState*NB_COL+state];
    else { ISOException.throwIt(ISO7816.SA_NO_SUCH_STATE); }
}
public void endStateMachine(short terminalState) {
    // checks if no successor
    for (short i = 0; i < NB_COL; i++) {
        if (securityAutomaton[currentState*NB_COL+i]
                            != NO_STATE)
            ISOException.throwIt(ISO7816.SA_NO_SUCH_STATE);
    } // then reinitialize the security automaton
    currentState = START_STATE_MACHINE; }
```

5 Conclusion

We have presented a general countermeasure in this paper, which can be applied to smart cards in order to detect FI attacks. The main idea was to provide redundancy of the control flow using a security automaton executed in the kernel mode. It allows to dynamically check the behavior of the program. We automatically generated the automaton during the linking process of the applet and for adding specific check points, we allow the developer to insert calls to a method `setState()`. For efficiency, we removed this call from the binary file and we replaced it by a simple `goto` which enforces the verification. In the second step, we applied this technique without modifying the JCVM by executing the transition functions in an API. We developed an analysis tool that checks the coherence of the security policy.

Of course, this technique is not limited to CFG properties but it can be used for more general security policy if they can be expressed as *safety properties*. In particular, it is interesting to check if some security commands have already been done before executing sensitive operation. Some are memorized in secured container (*i.e.* the PIN code field `isValidated`) but some of them use unprotected fields and could be subjected to FI attacks. The difficulty is to find the right trade off between a highly secured system with a poor performance or an efficient system with less security.

Acknowledgments. This work is partly funded by the French project INOSSEM (PIA-FSN2-Technologie de sécurité et résilience des réseaux).

References

1. Aktug, I.: Algorithmic Verification Techniques for Mobile Code. Ph.D. thesis, KTH, Theoretical Computer Science, TCS, qC 20100628 (2008)
2. Al Khary Sere, A.: Tissage de contremesures pour machines virtuelles embarquées. Ph.D. thesis, Université de Limoges (2010)

3. Anderson, R., Kuhn, M.: Low Cost Attacks on Tamper Resistant Devices. In: Christianson, B., Crispo, B., Lomas, M., Roe, M. (eds.) Security Protocols 1997. LNCS, vol. 1361, pp. 125–136. Springer, Heidelberg (1998)
4. Barbu, G.: On the security of Java Card platforms against hardware attacks. Ph.D. thesis, Grant-funded PhD with Oberthur Technologies and Télécom ParisTech (2012)
5. Barbu, G., Andouard, P., Giraud, C.: Dynamic Fault Injection Countermeasure A New Conception of Java Card Security. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 16–30. Springer, Heidelberg (2013)
6. Blömer, J., Otto, M., Seifert, J.P.: A new CRT-RSA algorithm secure against bellcore attacks. In: Computer and Communications Security, pp. 311–320 (2003)
7. 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)
8. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
9. Dubreuil, J., Bouffard, G., Lanet, J.L., Iguchy-Cartigny, J.: Type classification against Fault Enabled Mutant in Java based Smart Card. In: ARES 2012, pp. 551–556. IEEE, Prague (2012)
10. Farissi, I.E., Azizi, M., Moussaoui, M., Lanet, J.L.: Neural network Vs Bayesian network to detect javacard mutants. In: Colloque International sur la Sécurité des Systèmes d'Information (CISSE), Kenitra Marocco (March 2013)
11. Girard, P., Villegas, K., Lanet, J.L., Plateaux, A.: A new payment protocol over the Internet. In: CRiSIS 2010, pp. 1–6 (2010)
12. Joye, M., Quisquater, J.J., Bao, F., Deng, R.H.: RSA-type signatures in the presence of transient faults. In: Darnell, M.J. (ed.) Cryptography and Coding 1997. LNCS, vol. 1355, pp. 155–160. Springer, Heidelberg (1997)
13. McDougall, M., Alur, R., Gunter, C.A.: A model-based approach to integrating security policies for embedded devices. In: 4th ACM International Conference on Embedded Software, EMSOFT 2004, pp. 211–219. ACM, New York (2004)
14. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing High-Level Security Properties for Applets. In: Quisquater, J.-J., Paradinas, P., Deswarte, Y., El Kalam, A.A. (eds.) Smart Card Research and Advanced Applications. IFIP, vol. 153, pp. 1–16. Springer, Heidelberg (2004)
15. Prevost, S., Sachdeva, K.: Application code integrity check during virtual machine runtime (August 2004)
16. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.-L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Alcaraz Calero, J.M., Thomas, T. (eds.) SNDS 2012. CCIS, vol. 335, pp. 185–194. Springer, Heidelberg (2012)
17. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
18. Skorobogatov, S.P., Anderson, R.: Optical Fault Induction Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 31–48. Springer, Heidelberg (2003)
19. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
20. Wagner, D.: Cryptanalysis of a provably secure CRT-RSA algorithm. In: 11th ACM Conference on Computer and Communications Security, pp. 92–97 (2004)