Vulnerability Analysis on Smart Cards using Fault Tree

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

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

Abstract. In smart card domain, attacks and countermeasures are advancing at a fast rate. In order to have a generic view of all the attacks, we propose to use a Fault Tree Analysis. This method used in safety analysis helps to understand and implement all the desirable and undesirable events existing in this domain. We apply this method to Java Card vulnerability analysis. We define the properties that must be ensured: integrity and confidentiality of smart card data and code. By modeling the conditions, we discovered new attack paths to get access to the smart card contents. Then we introduce a new security API which is proposed to mitigate the undesirable events defined in the tree models.

Keywords: Smart Card, Security, Fault Tree Analysis, Attacks, Countermeasures

1 Introduction

A smart card is an intelligent and efficient device which stores data securely and also ensures a secure data exchange. Security issues and risks of attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. This requires clear understanding and analysis of possible ways of attacks and methods to avoid or mitigate them through adequate software and/or hardware countermeasures. To further understand the possible attack paths, a common method is required and in this paper we try to define a method which can help to sort these issues to an extent.

Often countermeasures are dedicated to an attack and do not take into account a global point of view. Usually designers use an inductive (bottom-up) approach where, for each attack a mechanism is added to the system to mitigate the attack. In such an approach, a probable event is assumed and the corresponding effect on the overall system is tried to be ascertained. On one hand, this allows to determine the possible system states from the knowledge of the attacker model and on the other hand the feasible countermeasures for the attack. Thus, it leads to several defenses where the final goal is not clearly defined nor ensured. Moreover the overhead of the defenses is high both in terms of memory footprint and CPU usage.

In this paper, we suggest to adopt a deductive approach (top-down) by reasoning from a more general case to a specific one. We start from an undesirable event and an attempt is made to find out the event, and the sequence of events which can lead to the particular system state. From the knowledge of the effect, we search a combination of the causes. We postulate a state, in our case a security property, and we systematically build the chain of basic events.

This approach is based on a safety analysis, often used for safety critical systems. The safety analysis performed at each stage of the system development is intended to identify all possible hazards with their relevant causes. Traditional safety analysis methods include, *e.g.* Functional Hazard Analysis (FHA) [1], Failure Mode and Effect Analysis (FMEA) [2] and Fault Tree Analysis (FTA). FMEA is a bottom-up method since it starts with the failure of a component or subsystem and then looks at its effect on the overall system. First, it lists all the components comprising a system and their associated failure modes. Then, the effects on other components or subsystems are evaluated and listed along with the consequence on the system for each component's failure modes. FTA, in particular, is a deductive method to analyze system design and robustness. Within this approach we can determine how a system failure can occur. It also allows us to propose countermeasures with a higher coverage or having wider dimension.

This paper is organized as follows, section 3 is about Java Card security. Section 4 describes smart card vulnerability analysis using FTA. Section 5 presents an API to mitigate undesirable events. Conclusions and future works are summarized in section 6.

2 Using FTA for security analysis: Related Works

FTA is often used for reliability analysis but it can be also used for computer security. In [3], the authors suggested to integrate FTA to describe intrusions in an IT software and Colored Petri Net (CPN) to specify the design of the system. The FTA diagrams are augmented with nodes that describe trust, temporal and contextual relationships. They are also used to describe intrusions. The models using CPNs for intrusion detection are built using CPN templates. The FTA restricts drastically the possible combination of the events. In [4], FTA is used to assess vulnerability considering the fact that the undesirable events of interest should already be in a fault tree designed for the purpose of a risk assessment. They showed how to build specific FTA diagrams for vulnerability assessments. In [5], an attack tree is used to model potential attacks and threats concerning the security of a given system. Attack trees are always having the same meaning as FTA (same gate connectors). They generate attack scenario in a close way to Unified Modeling Language (UML) scenario for evaluating, for example, a buffer overflow in the system. Another work [6] described a methodology to analyze both software and hardware failure in a system and also discussed the use of FTA affordable for software systems by design, *i.e.* incorporating countermeasures so that the fault can be considered and mitigated during analysis. Byres *et al.* [7] used the BDMP[1] model to find vulnerabilities in SCADA system.

3 Security of Java based smart cards

Java Card is a kind of smart card that implements the standard Java Card 3.0 [8] specification. Such a smart card embeds a Virtual Machine (VM) which interprets codes already romized with the operating system or downloaded after issuance. Java Cards have shown an improved robustness compared to native applications with respect to many attacks. Java Card is an open platform for smart cards, *i.e.* able to load and execute new applications after issuance. For security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [9]. This protocol ensures that the owner of the code has the necessary credentials to perform the action. Thus different applications from different providers can run on same smart card. Using type verification, the byte codes delivered by the Java compiler and the converter (in charge of delivering compact representation of Class files name Cap file) are protected, *i.e.* and the application loaded is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications on the card, enforcing isolation between them. Until now it was safe to presume that the firewall was efficient enough to avoid malicious applications (applet modified after off-card verification). Nevertheless some attacks have been successful in retrieving secret data from the card.

Smart cards are designed to resist numerous attacks using both physical and logical techniques. There are different ways to attack Java Card and many of them are associated with countermeasures. An attacker can use physical or logical attacks [10,11] or combined attacks [12,13] (combination of physical and logical). Often ad-hoc countermeasures can only protect the card from a single type of attacks. So if a manufacturer wants to protect his card against different types of attacks he must embed several countermeasures. Here we are presenting a model to avoid embedding unwanted countermeasures which in turn helps to protect the embedded system against attacks.

4 Smart card vulnerability analysis using FTA

4.1 Introduction

FTA is an analytical technique (top-down) where an undesirable event is defined and the system is then analyzed to find the combinations of basic events that could lead to the undesirable event. A basic event represents an initial cause which can be a hardware failure, a human error, an environmental condition or any event. The representation of the causal relationship between the events is given through a fault tree. A fault tree is not a model of all possible system failures but a restricted set, related to the property evaluated.

[1] Boolean logic Driven Markov Process

FTA is a structured diagram constructed using events and logic symbols mainly AND and OR gates (Fig. 1). Other logical connectors can be used like the NOT, XOR, conditional etc. The AND gate describes the operation where all inputs must be present to produce the output event. The OR gate triggers the output if at least one of the input events exists. The INHIBIT gate describes the possibility to forward a fault if a condition is satisfied. For readability of FTA diagrams, a triangle on the side of an event denotes a label, while a triangle used as an input of a gate denotes a sub tree defined somewhere else (Fig. 5).

In our work, we define four undesirable events that represent the failure of the smart card system: code or data integrity, code or data confidentiality. Code integrity is the most sensible property because it allows the attacker to execute or modify the executable code which not only leads to the code and data confidentiality but also the data integrity. The Java Virtual Machine (JVM) prevents these events to occur under normal execution. Intermediate nodes in the tree represent a step in an attack while leafs (basic events) represent either the state of the system or a known attack. If this attack needs a given state of the system to succeed then an AND gate must bind the attack to the state of the system. Thus it becomes a condition on the system state. For example, the EMAN 2 [13] attack shown in Fig. 2, modifies the return address. However, it is necessary to get access to the content of the return address which is represented by the intermediate node on the top. For modifying the return address, the attacker must modify the CAP file and thereby the VM allows the execution of ill typed applet. This latter condition is true if all the basic events are true: absence of an embedded Byte Code Verifier (BCV), no run time check of the index of the local variables and absence of integrity check of the system context area in the frame. The BCV ensures that the applet to be installed is compliance with the Java Card security rules. The presence of one of these countermeasures is enough to mitigate the attack.

4.2 Code Integrity

The first property to be analyzed in a smart card for understanding or implementing security features is the code integrity. If the attacker is able to modify the method area or more generally to divert the control flow of the code to be executed, then he will have the possibility to read or modify the data and code stored inside the card. We consider two possibilities here, see Fig. 3, either the processor executes a new code (execution of an arbitrary shell code) or it executes the regular code in such a way that the initial semantics is not preserved. In the left part of the tree (execution of a shell code), we can obtain it either by changing the control flow by modifying of the branch condition [13] or the return address as explained in the previous section. In the right branch of the fault tree diagram, we modify the execution flow just like a program counter execute an existing code. These attacks refer to code mutation were operands can be executed instead of instructions as explained in [14]. In this class of attack, we can find the attacks mentioned in [15] concerning the Java exception mechanism, the attacks are not only related to the modification of the jump offset but also

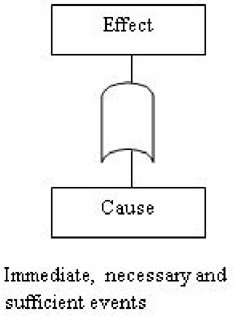

Fig. 1. The causal relationship between events

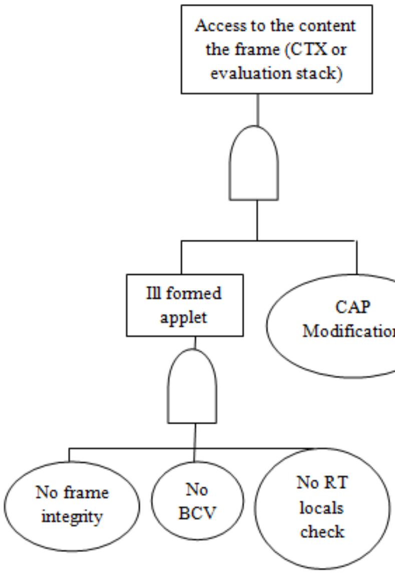

Fig. 2. Steps for an Eman 2 attack

through a Java Program Counter (`jpc`) corruption. Then all these intermediate nodes need to be refined to determine all the causes of the unwanted events.

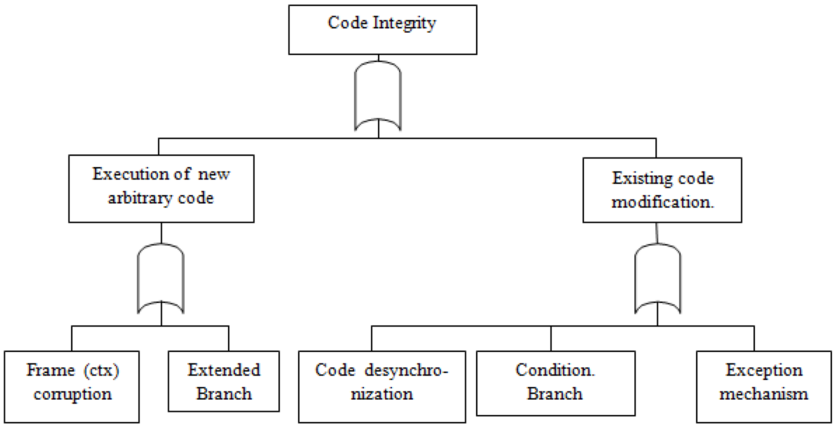

Fig. 3. Code integrity tree

At that step we need to find the most efficient countermeasure to protect the code integrity. As described previously, each attack can be mitigated through a dedicated countermeasures, e.g. for the Eman 2 attack, it can be either through the return address stored in a dedicated place (not accessible through a local variable), a frame integrity mechanism, an embedded verifier, verification of the indices of the locals during the load time or some run time checks. One can remark that if the countermeasure is closed to the root of the tree its efficiency in terms of coverage will be higher which doesn't mean that the countermeasure must be close to root. There are other parameters to be taken into account like the run time cost, the latency, the memory footprint etc. For example the Dynamic Syntax Interpretation (DSI) has a high coverage as shown in Fig. 4: this countermeasure is at the top of the tree. A lot of other countermeasures can also be applied here. This work has been extended to three other undesirable events.

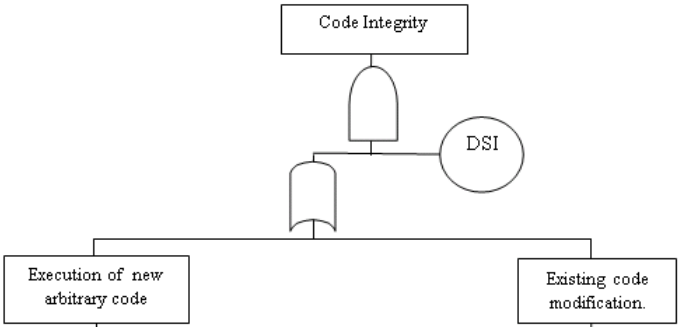

Fig. 4. An efficient countermeasure for code integrity

4.3 Basic Events: Possible Attacks

Known attacks are considered as basic events. Several authors proposed and tested different types of attacks which are discussed here.

Abusing shareable interface is an interesting technique to trick the virtual machine (VM). The principal mode of shareable interface attack [10] is to obtain type confusion without modifying the Cap files. To accomplish such an attack, the authors created two applets which are communicated using the shareable interface mechanism. Each applet uses a different type of array to exchange data in order to create the type confusion. In this case, the BCV cannot detect any problem during compilation or while loading the files. In our experience, most of the cards with an on-card BCV refused to allow applets using shareable interface mechanism. Since it is impossible for an on-card BCV to detect this kind of abnormality, Hubbers *et al.* [10] proposed the hypothesis of shareable interface.

Another type of attack is the abort transaction, which is very difficult to implement but it is a widely used concept in database system. The purpose of this transaction is to make a group of atomic operations. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction and reset the references of such objects to null. However, the authors found that some cases where the card keeps the reference to objects allocated during transaction, even after a rollback.

A fault model is explained in [16,17] to attack the smart card by physical means. Nowadays, it is also possible to obtain precise byte errors using a laser beam injection into the smart card memory or to perturb the runtime execution (like a pipeline modification). Due to this physical fault, a logical attack can occur which is called a combined attack. Barbu *et al.* proposed an attack based on the `checkcast` instruction [15] which is used to modify the runtime type by injecting a laser beam. Then this applet gets installed on the card after it has been checked by an on card BCV and it is considered to be a valid applet. The aim of this attack is to create a type confusion to forge a reference of an object and its contents. The authors also explained the principle of instance confusion, similar to the idea of type confusion where the objective is to confuse an instance of object A to an object B by inducing a fault dynamically by using a laser beam.

Bouffard *et al.* described in [13], two methods to change the control flow graph of a Java Card. The first one is EMAN 2, which provides a way to change the return address of the current function. This information is stored in the Java Card stack header. Once the malicious function exits during the correct execution, the program counter returns to the instruction which addresses it. The address of the JPC is also stored in the Java Card Stack header. An overflow attack success to change the return address by the address of the malicious byte code. Since there is no runtime check on the parameter, it allows a standard buffer overflow attack to modify the frame header.

4.4 Basic Events: Possible Countermeasures

For stack area protection, Girard mentioned in [18], that the system area of the frame is very sensible. So, he suggested to split the frame and manage the stack context separately from the stack and the locals of the frame. In [15], Barbu proposed a real approach by modifying the value of the security context in case of an illegal modification delegating the control to the firewall. He also suggested a stack invariant by XOR-ing the pushed and popped value. Dubreuil *et al.* [14] proposed to use a typed stack without paying the cost of two stacks manipulation. In the case of method area protection, it is possible to execute data as program, considering the value to be XOR-ed should be dependent to the type of the variables as suggested in [18]. Here, the data are protected within the same bounds of possibility. Barbu [15] proposed exactly the same approach, while Razafindralambo *et al.* [19] demonstrated that it is possible to recover the XOR value by using a dynamic syntax adding randomization to the XOR function. The implementation of all these countermeasures are during the linking step. In order to avoid the exploitation of the linker, an algorithm has been proposed in [14] to efficiently check the usage of the token to be linked by manipulating the reference location component.

Séré proposed three countermeasures in [20]. The first one is the bit field method based on the nature of the byte code that must remain non mutable (*i.e.* an instruction cannot be replaced by an operand). The second one is the basic block method, which is an extension of the work described in [4], where a checksum is calculated during the execution of a linear code fragment and it is checked during the exit. The last one is the path check method, a completely different approach based on the difference between the expected execution paths and the current one. Any divergence or code creation can be detected by coding the path in an efficient way.

These countermeasures are of particular interest because they are addressing the two events of the first level of code integrity. The patent [21] describes a similar approach to the third countermeasure of Séré, but it is performed at the applicative level. A similar approach, based on flags and preprocessing is there to detect unauthorized modification in the control flow, as mentioned in [22]. To avoid the exploitation of the `checkcast` attack, an efficient implementation of this instruction was proposed in [15].

5 Definition of an API to Mitigate the Undesirable Events

Countermeasures can be implemented at two levels: virtual machine (system) or an applicative. The main advantage with system countermeasures is that the protections are stored in the ROM memory, which is a less critical resource than the EEPROM. So it is easier to deal with integration of the security data structures and code in the system. With applicative countermeasures, it is possible to implement several checks inside the application code to ensure that the program always executes a valid sequence of code on valid data. It includes double condition checks, redundant execution, counter, etc. and are well known by the developers. Unfortunately, this kind of countermeasures drastically increase the program size because, besides the functional code, it needs security code and the data structure for enforcing the security mechanism embedded in the application. Furthermore, Java is an interpreted language therefore its execution is slower than that with a native language. So, this category of countermeasures experiences bad execution time and adds complexity for the developer. But the main advantage here is that the developer has a precise knowledge of the sensitive part of his code. A new approach exists, which is driven by an application and it includes means in the system layer.

This solution presents a good balance between memory footprint and optimization of the countermeasure. This also presents a possibility to offer security interoperability between several manufacturers who implement this API.

5.1 Model of attacks

Using the FTA analysis, we defined attack paths that allows a simple laser based fault injection. They correspond to the effect of a laser fault on the JVM. By refining the analysis, we discover three new possibilities:

- If the effect of laser affects the value returned by a method it can execute a code fragment without the adequate authorization or in a conditional evaluation it can change the branch which is to be executed. This attack targets the Java Card evaluation stack where the value returned by the function is stored,
- The effect of fault can also bypass the execution of a given method. By modifying the JPC one can jump over a method that performs a security test letting on top of the stack the address of the called object. Of course, the stack has a high probability to be ill typed (non compatible) at the end of the current method. Nevertheless an attacker can still send high value information before being detected,
- It can also modify the address of data to be copied in the I/O buffer or change the number of byte to be copied. This attack allows to dump memory in an arbitrary way.

They have been introduced as a new intermediate node: JPC CORRUPTION and a new basic event: EVALUATION STACK CORRUPTION which can be obtained either by a logical attack or a fault attack. They can be found not only on different branches of the code integrity tree but also on the code confidentiality tree like the leafs of the intermediate node CONDITIONAL BRANCH, they are combined with an AND condition to generate the CODE DE-SYNCHRONIZATION intermediate node.

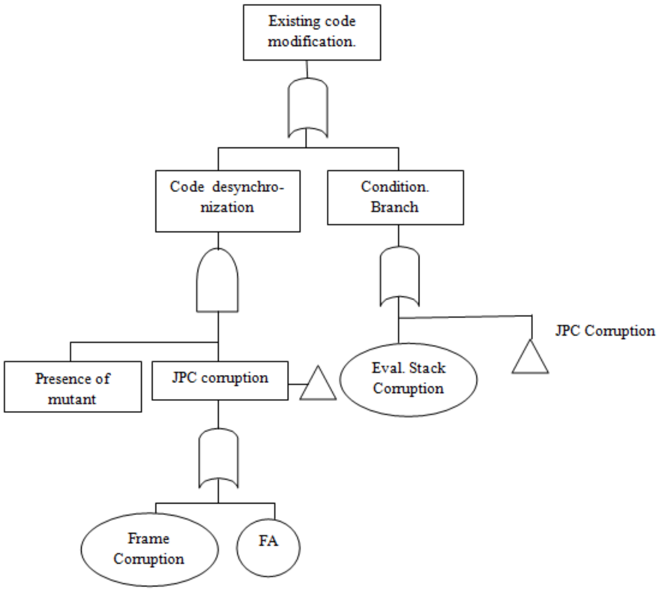

Fig. 5. JPC corruption and its effects

With the FTA analysis we were able to determine new potential vulnerabilities. The next step is to design efficient countermeasures to ensure a high coverage with a minimal memory footprint.

5.2 The INOSSEM API

Since several attacks can be performed against smart cards, an API has been defined, so that a given application can request to increase security for a code fragment. For example, to mitigate all the attacks against the stack, (*i.e.* the parameters of a called method), it is possible to invoke a method with a signature added as a parameter. Then, the application at the beginning of the execution of the called method should verify that the parameters have not been modified by a fault attack (listing 1.1).

```
// initialize the signature object with the first parameter
paramChkInstance.init(firstParam);
paramChkInstance.update(secondParam);//update signature buffer
short paramChk = paramChkInstance.doFinal(userPIN); // sign it
// invoke the sensitive method
sensitiveMeth(firstParam ,secondParam ,userPIN ,paramChk);
```

Listing 1.1. Use of the `ParamCheck` class to sign parameters

All sensitive methods of the Java Card applet must ensure that no attack against the integrity of the frame occurred during the call. If the signature does not match, the card must be blocked, as shown in the listing 1.2

```
void sensitiveMeth (byte p1,short p2,OwnerPIN p3,short chk) {
// initialize the signature object with the parameters
paramChkInstance.init(p1); paramChkInstance.update(p2);
//... check the integrity of the locals in the frame
if (paramChkInstance.doFinal(p3) != chk)
    {// something bad occurred take some action }
```

Listing 1.2. Use of the `ParamCheck` class to verify parameters.

The second attack is related with JPC corruption. The control flow of the program can be modified due to the attack. For mitigating it, we choose to implement a security automaton into the API. The security automaton represents the security policy in terms of control flow checks. It is a partial representation of the control flow of the program, thus providing a redundancy that can be checked during the execution of the applet. For example, before decrementing the value of a counter the authentication step must have occurred earlier. So we included some states' modification into the code with a call to `setState()` method of the API. In the constructor of the applet, we have initialized a structure representing the security automaton. Each time a call to `setState()` occurs, the system checks if the current state allows the transition to the new state, if not it throws a security exception (listing 1.3).

```
private void debit(APDU apdu) {
this.setState(CRITICAL_SECTION);// transition to a new state
if (!pin.isValidated()) {//disallows if PIN is not verified
    this.endStateMachine(PIN_VERIFICATION_REQUIRED_STATE);
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);}
```

Listing 1.3. The security automaton

There are also Java classes which are used to protect local variables against integrity and confidentiality violation. These form a coherent means to protect the application against the attack that the FTA analysis brought to the fore.

6 Conclusion and Future Works

A Fault Tree Analysis method was proposed and applied in the smart card domain which can be used for safety and vulnerability analysis. We have identified four major undesirable events and refined the analysis to reach till the basic events representing the effect of either laser attacks, logical attacks or their combination. During the analysis we brought forth three new attacks considering the effect of a single laser fault. And in the INOSSEM project the partners defined an API that provides protections against these attacks. With this strategy, a smart card developer can adapt the level of security of the JVM for a given code fragment of its applet. We are currently considering quantification of the probability for an attacker to reach his objective in a given time or the overall mean time for the attack to overcome it. This on going work uses the BDMP formalism [23] with the models designed in this paper.

7 Acknowledgements

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

References

1. Leveson, N.G.: Software Safety - What, Why And How? ACM Computing Surveys 16(2) (1986) 125–164
2. DH, S.: Failure Mode and Effect Analysis: FMEA from Theory to Execution. ASQ Press (1995)
3. Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L., Wang, Y., Lutz, R.: Software fault tree and colored petri net based specification, design and implementation of agent-based intrusion detection systems. IEEE Transactions of Software Engineering, submitted (2002)
4. Prevost, S., Sachdeva, K.: Application code integrity check during virtual machine runtime. (2004) US Patent App. 10/929,221.
5. Moore, A.P., Ellison, R.J., Linger, R.C.: Attack modeling for information security and survivability. Technical report, DTIC Document (2001)

6. Fronczak, E.: A top-down approach to high-consequence fault analysis for software systems. In: PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, IEEE (1997) 259
7. Byres, E.J., Franz, M., Miller, D.: The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems. In: International Infrastructure Survivability Workshop (IISW'04), IEEE (2004)
8. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2011)
9. GlobalPlatform: Card Specification. 2.2.1 edn. GlobalPlatform Inc. (2011)
10. Hubbers, E., Poll, E.: Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen (2004)
11. Iguchy-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. Journal in Computer Virology **6**(4) (2009) 343–351
12. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Smart Card Research and Advanced Application. Volume 6035 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 133–147
13. Bouffard, G., Lanet, J.L., Iguchy-Cartigny, J.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Smart Card Research and Advanced Applications. Volume 7079 of Lecture Notes in Computer Science., Berlin / Heidelberg, Springer (2011) 283–296
14. Dubreuil, J., Bouffard, G., Lanet, J.L., Iguchy-Cartigny, J.: Type classification against Fault Enabled Mutant in Java based Smart Card. In: Sixth International Workshop on Secure Software Engineering (SecSE), Springer (2012) 551–556
15. Barbu, G.: On the security of Java Card$^{\mathrm{TM}}$platforms against hardware attacks. PhD thesis, Grant-funded with Oberthur Technologies and Télécom ParisTech (2012)
16. Blömer, J., Otto, M., Seifert, J.P.: A new CRT-RSA algorithm secure against bellcore attacks. In: ACM Conference on Computer and Communications Security, Washington, DC, USA, ACM (2003) 311–320
17. Wagner, D.: Cryptanalysis of a provably secure CRT-RSA algorithm. In: ACM Conference on Computer and Communications Security, Washington, DC, USA, ACM (2004) 92–97
18. Girard, P.: Contribution à la sécurité des cartes à puce et de leur utilisation. Habilitation thesis, University of Limoges (2011)
19. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Computer Networks and Distributed Systems Security. Volume 335 of Communications in Computer and Information Science., Springer (2012) 185–194
20. Al Khary Séré, A.: Tissage de contremesures pour machines virtuelles embarquées. PhD thesis, Université de Limoges, 123 Avenue Albert Thomas, 87100 Limoges Cedex (2010)
21. Akkar, M.L., Goubin, L., Ly, O., et al.: Automatic integration of counter-measures against fault injection attacks. Pre-print found at http://www.labri.fr/Perso/ly/index.htm (2003)
22. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) **13**(1) (2009) 4
23. Pietre-Cambacedes, L., Bouissou, M.: Attack and Defense Modeling with BDMP. In: MMM-ACNS. Volume 6258 of Lecture Notes in Computer Science., Springer (2010) 86–101