Mitigating Type Confusion on Java Card

Jean Dubreuil, Smart Secure Devices (SSD) Team, University of Limoges, Limoges Cedex, France

Guillaume Bouffard, Smart Secure Devices (SSD) Team, University of Limoges, Limoges Cedex, France

Bhagyalekshmy N. Thampi, Smart Secure Devices (SSD) Team, University of Limoges, Limoges Cedex, France

Jean-Louis Lanet, Smart Secure Devices (SSD) Team, University of Limoges, Limoges Cedex, France

ABSTRACT

One of the challenges for smart card deployment is the security interoperability. A smart card resistant to an attack on a given platform should be able to guarantee the same behavior on another platform. But the current implementations do not comply with this requirement. In order to improve such standardization the authors propose a framework based on annotations with an external pre-processing to switch the Java Card Virtual Machine (JCVM) into a secure mode by activating a set of countermeasures. An example has been proposed in this paper for implementing a countermeasure against type confusion with a fault attack. Smart cards are often the target of software, hardware or combined attacks. In recent days most of the attacks are based on fault injection which can modify the behavior of applications loaded onto the card, changing them into mutant applications. This countermeasure requires a transformation of the original program byte codes which remain semantically equivalent. It needs a modification of the JCVM which stays backward compatible and a dedicated framework to deploy these applications. Thus, the proposed platform can resist to a fault enabled mutant.

Keywords: *Countermeasures, Faults, Java Card, Smart Card, Typed Stack, Viruses*

1. INTRODUCTION

Owing to NFC technology, the usage of mobile phone for e-transaction will increase drastically in upcoming years. This transition in technology requires more applications to be developed in mobile phones and some of them are highly sensitive and need high level security. For example, nowadays using mobile phones for payment, ticketing, mobile TV, *etc.* are common. Thus, it raises the question of confidence of the terminal that delivers these services. If a platform is multi-applicative, then loading a new application can reduce the security of already installed application. Through virtualization, the platforms have reached interoperability

DOI: 10.4018/jsse.2013040102

at functional level and it will be difficult to obtain it at security level. In case of this security interoperability property has not been achieved, then the consequence will imply a costly certification process for each application, for all configurations of the application. This could be a main point for not adopting NFC applications and thereby reducing the development's effort for mobile transactions. Critical applications in a mobile phone should be hosted by a Secure Element (SE) often implemented with a smart card.

Mobile Network Operators (MNO) are looking to open their native SE: the Universal Subscriber Identity Module (USIM) Card (ESTI, 2005) to the third party service providers, in order to allow them to develop value added services, like m-payment or m-transport *etc.*, on NFC based technology. To warrant security for third party applications hosted in USIM Card, MNO have chosen to certify their SE under the Common Criteria (CC) (The Common Criteria for Information Technology Security Evaluation) scheme (Common Criteria Organization, 2012). This will facilitate post issuance for third party applications downloading without existing CC certification loss which means an industrial process will have to be in place, which has to warrant the innocuousness of every candidate's application for a download. In particular, an application certified on a given platform must have the same behavior in other platforms also.

Nowadays most of the USIM cards are based on a Java Card Virtual Machine (JCVM). Java Card is a type of smart card that implements the standard Java Card 3.0 (Sun, 2010) in one of the two editions "Classic Edition" or "Connected Edition". Such a smart card embeds a Virtual Machine (VM), which interprets application byte codes already *romized* with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform (Global Platform, 2011). This protocol ensures that, the code owner has the necessary credentials to perform the particular action.

Java Cards have shown an improved robustness compared to native applications concerning many attacks. They are designed to resist numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and especially fault attacks. A fault attack modifies a part of the memory content or signal on internal bus and leads to deviate from the exploitable behavior, not by an attacker. A comprehensive consequence of such attacks is mentioned in Iguchi-Cartigny and Lanet (2010). Although fault attacks have been generally used in the literature from a cryptanalysis point of view (Aumüller, Bier, Fischer, Hofreiter, & Seifert, 2003; Hemme, 2004; Piret & Quisquater, 2003), they can also be applied to every code layer embedded in a device. For example, by choosing the exact byte of a program, an attacker can bypass logical tests. To avoid such attacks, several countermeasures have been designed to protect the execution flow, the integrity of Java fields, the confidentiality of the byte code, *etc*.

Designing efficient countermeasures against fault attacks are important not only for smart card manufacturers but also for application developers. Manufacturers need countermeasures with the lowest cost in terms of memory and processor usage. The coverage (reduction of the number of succeeding attacks) and the detection latency (number of instructions executed between an attack and its detection) are the most important metrics in the development process of smart card. In order to minimize the impact of fault attacks, developers need to implement countermeasures in their applicative code. Examples of such applicative countermeasures are redundant branch instructions, counters, redefining the value of true and false constants, *etc*. But in this case the developer should have the knowledge of underlying platform architecture, which can differ from each smart card model. This low interoperability of the security aspects between different platforms is a huge problem for smart card application developments and certifications.

The proposed solution in this paper adapts a security feature found in the Java Card 3 platform annotations which is also fully applicable to Java Card 2 platform using a pre-processor. When the VM interprets the application code and enters a method or class tagged with a security annotation, it switches to a "secure mode". We propose here an efficient countermeasure that follows this approach and which is able to protect the VM against type confusion.

The contribution of this paper with respect to our prior work is based on two mechanisms: a novel system countermeasure based on JCVM verification of the Java element type, and a framework to adapt the Java byte code to the proposed countermeasure.

This paper is organized as follows; the first section provides a brief state of the art on smart cards attacks, the existing countermeasures and the impact of the fault and mutant generation. The second section introduces the developed countermeasure. The evaluation framework and the collected metrics are highlighted in the third section and finally, section four concludes the work with the future perspectives.

2. ATTACKS AND COUNTERMEASURES

A Java Card attack can be carried out in two ways.

The first one is a logical attack in which an attacker uses an ill–formed applet to obtain sensitive information stored in the card. For obtaining it, the applet will try to execute some illegal instructions to read and write in the smart card memory as explained in Iguchi-Cartigny and Lanet (2010). This can be accomplished by making a type confusion attack or by changing the control flow graph (CFG). Type confusion blurs the Java Card Runtime Environment (JRE) to use reference to an object's instance as a value. In Java Card, references are mainly stored as 16-bit, *i.e.* the size of a short. This attack can be achieved by pushing a value and manipulating it as a reference (and *vice versa*). It offers the ability to manipulate pointers; even though Java security model forbids the use of pointers. Indeed, Java is a strong typed language, thus it is illegal to perform arithmetic operations on reference. Some of the recent smart cards with more resources can include a Byte Code Verifier (BCV). During the installation step, this BCV checks and prevents to install incorrect and malicious applets.

The second one is attacking Java Card by modifying the physical layout.

2.1. Fault Attacks

Faults can be injected into the chip by inducing perturbations in its execution environment (Bar-El, Choukri, Naccache, Tunstall, & Whelan, 2006). Faults can also be injected by some physical attacks which expose the device to some sort of physical stress. As a result, the device has erratic behaviour, *i.e.*, changing values in memory cells, transmitting different signals through bus lines, or damaging the structural elements. Thus, these errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. These perturbations can have various effects on the chip registers (program counter, stack pointer), or on the memories (variables and code changes). Mainly, it can permit an attacker to execute a treatment beyond his rights, or to access secret data in the smart card. Fault attack is an old research field mainly in avionics or space domains (Ziegler et al., 1996). Researchers brought to the fore that cosmic rays can flip single bits in the memory of an electronic device. Such faults are still an issue until now for those devices. Three types of fault attacks are focused by researchers in the smart card field like power spikes, clock glitches and optical attacks.

A smart card is a portable device without embedded power supply or clock and thus it requires a smart card reader (which provides external power and clock sources) for operating it. The reader can also be replaced by an attacker with specific equipment in the laboratory. Short variations of the power supply can induce errors into the smart card internal operations. Spikes not only allow injecting memory faults but also faults in the execution of a program. To confuse the program counter can make conditional checks to work improperly, loop counters to be decreased and arbitrary instructions to be executed.

The reader provides to the card a clock signal, which may incorporate short deviations beyond the required tolerance from the standard signal bounds. Such signals are called glitches. They can be defined by a range of different parameters and can be used to inject memory faults as well to generate faulty execution behaviours. Hence, the possible effects are the same as in 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. These memory cells, *i.e.*, EEPROM and semiconductor transistors, have been found to be sensitive to light. This occurs thanks to photoelectric effect. Modern green or red lasers can be focused on relatively small regions of a chip, such that faults can be targeted fairly well. Another method is to make changes in the external electrical field of the smart card and it has been considered as a possible method for inducing faults.

2.2. Fault Model

To prevent a fault attack from being occurred, it is necessary to know its effects on smart cards. Fault models have already been discussed in details (Blomer, Otto, & Seifert, 2003; Wagner, 2004). The existing fault models, given in descending order in terms of attacker's power are shown in the Table 1. An attack using the precise bit error model had been discussed in Skorobogatov and Anderson (2003). But it is not realistic on current smart cards as modern components implement hardware security on memory like error correction and detection code or memory encryption. Barbu *et al.*, in 2010, use a precise byte errors model. To have a precise location, they have a white box model on the attacked Java Card. The unknown byte errors model is motivated by the fact with the attacker's power is effectively reduced by the targeted memory encryption and, on some cards, a randomized clock. In this case, the attacker knows this attack is a success but he has not knowledge about the position of the block as it is used in the CPU. Finally, high-secured smart cards are armed with countermeasures (encrypted memory, scrambled address and a randomized clock). These countermeasures imply that any error induced into the RAM, EEPROM or CPU at an undetermined moment give at most the information that a certain variable is faulty as explained in Blomer, Otto, and Seifert (2003).

In fact, an attacker injects physically energy into a memory cell to switch its state. According to the underlying technology, the memory will physically takes the value `0x00` or `0xFF`. If memories are encrypted, the physical value becomes a random value (more

Table 1. Existing fault model

Fault Model	Precision	Location	Timing	Fault Type	Difficulty
Precise bit errors	Bit	Precise control	Precise control	BSR[1], Random	++
Precise byte errors	Byte	Loose control	Precise control	BSR, Random	+
Unknown byte errors	Byte	Loose control	Loose control	BSR, Random	-
Random errors	Variable	No control	Loose control	Random	--

precisely a value which depends on the data, the address, and an encryption key). To be as close to the reality, we choose the precise byte error that is the most realistic fault model. Thus, we have assumed that an attacker can:

- Make a fault injection at a precise clock cycle (he can target at any operation he wants), only set or reset a byte to `0x00` or to `0xff` according to the underlying technology (BSR fault type), or he can change a byte to a random value beyond his control (random fault type);
- Target any memory cell he wishes (a specific variable or register).

Currently this accepted fault model is done by laser hits during the execution of a smart card command.

2.3. Security Mechanisms

Since a long time, smart card manufacturers have been aware of the danger of fault attacks. Hence, they have developed a large variety of hardware countermeasures (Ko, 2005). Major hardware countermeasures are sensors and filters, which aim to detect attacks, *e.g.*, using anomalous frequency detectors, anomalous voltage detectors, or light detectors. Other countermeasures use redundancy, *i.e.*, dual-rail logic (keeping data in two redundant memories), and dual hardware (computing a result twice in parallel). A data is considered to be error-free if both values (computed or memorized) match. But these are very expensive countermeasures, and hence, redundancy is not often implemented in smart cards.

We can notice that using only hardware countermeasures have two drawbacks. Highly reliable countermeasures are very expensive and low cost countermeasures only detect specific attacks. Since new fault attacks are being developed frequently these days, detecting only current known forms of physical tampering is not sufficient, especially for long term applications (an e-passport must be valid for 10 years).

An alternative or additional countermeasure is the use of software countermeasures. They are introduced at different stages of the development process. Their purpose is to strengthen the application code against fault injection attacks. Current approaches for software countermeasures include checksums, randomization, masking, variable redundancy, temporal redundancy and counters.

2.4. Applicative Countermeasure

Usually, it is the programmer who is in charge of adding defensive code to avoid any fault attacks. Generally this class of countermeasure produces application with a greater size. Hence, besides the functional code (the code that process data), we have the security code and the data structure for enforcing the security mechanism embedded in the application. Java is an interpreted language therefore it is slower to execute than a native language, so this category of countermeasures suffers from bad execution time and add complexity for the developer. Examples of such applicative countermeasures are: redundant if structure, step counters, loop counters, constant time execution, redundant variable (if possible with complementary value), specific coding of Boolean value, *etc.*

The following code is an abstraction of the Sun wallet example (Sun, 2010). According to the received command in the APDU, the user requests either a credit or a debit. The balance is protected with integrity which is checked with the method `readBalance`. Then before entering to either an increment or a decrement, the applet checks if the user has been previously authenticated. If not, it throws an exception. So, for the unauthenticated user, the security problem concerns the possibility to access the `incBalance` if not previously authenticated. The laser fault can change the `jpc` such that after reading the value of the balance it jumps to `incBalance` avoiding the authentication test. The minimal countermeasure is the step counter. The counter is initialized at its initial value and each time the control enters in a method it is decremented. Then in a sensible method (*e.g.* `incBalance`), the verification consists in checking that it passed to each decrementing step. Of course, the counter step is duplicated to avoid an attack on it and the both counter are checked.

Protecting RAM in a card is much easier than protecting EEPROM memory. So at the beginning, both counters are transferred in a transient memory and the modification of both counters must be protected by the Java Card transaction mechanism to avoid a smart card tear down (the two last points are not represented in the code)(see Listing 1).

To improve the counter mechanism, we can implement the state machine which checks if the next block is an authorized one by verifying that a call to a method is a valid one with regard to the current method.

The representation of the state machine uses an adjacency list which is a data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, a list of all other vertices which it has an edge to (that vertex's "adjacency list"). We can use a two-dimensional array which must be simulated in a Java Card (arrays are only one dimension in Java Card). Then the state machine SM is represented by SM = {{1, 2}, {3, 4}, {5, 6}, {}, {}, {}, {}}. (Figure 1)

Only four traces are allowed:

t1= {process, credit, readBalance, checkAuthentication, incBalance};

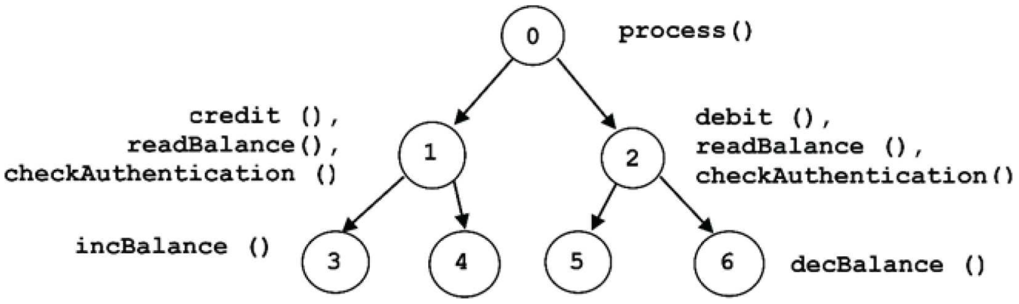
t2= {process, credit, readBalance, checkAuthentication, ISOException.throwIt};

t3= {process, debit, readBalance, checkAuthentication, decBalance};

Listing 1. The purse example

```
process (APDU apdu) {
   step = MAX;
   notStep = MIN;
   if (condition)
      debit(apdu);
   else
      credit (apdu);
}
credit (APDU apdu) {
   step -= DEC;
   notStep += DEC;
   short bal = readBalance();
   if (checkAuthentication())
      incBalance(bal);
   else
      ISOException.throwIt();
}
debit(APDU apdu) {
   short bal = readBalance();
   if (checkAuthentication())
      decBalance(bal);
   else
      ISOException.throwIt();
}
```

Figure 1. The control flow graph of the application



t4= {process, debit, readBalance, checkAuthentication, ISOException.throwIt}.

So, the program is transformed with some primitives, startStateMachine, which verifies that the last state was a leaf and reinitialize the state machine, `setState` which checks that the next state is allowed according to the current state. ST is hardcoded as a static final array (see Listing 2).

Adding security is not a so obvious process which needs a deep understanding of the hardware, the state of the art of the attacks and the most adequate countermeasure to use. The developer knows the assets of the program to be protected, but he needs to be aware of the attacks and the particular effects of fault attacks. For that reason, it is better to rely on system countermeasure embedded into the VM. The developer should indicate to the VM the variables which need a secure storage or the piece of codes which need a secure execution. For that, the developer also requires to signal to the VM the countermeasures which need to be enabled during a specific duration. Such mechanism will be described in the next section.

2.5. System Countermeasure

The objective of a system countermeasure is to detect an attack which occurs at linking time, run time (*e.g.* when the byte code transits on the data bus) or during the execution of another piece of code. Thus, the nature of the countermeasure is different in terms of:

- Protection of variable integrity: instance field, code to be executed, evaluation stack, execution context, *etc.*;
- Protection against control flow execution modification: bypassing a test, jumping to an unauthorized data area, jumping to an argument instead of an instruction, *etc.*;
- Execution of shell code;

Listing 2. The purse example with the state machine commands

```
process (...) {      credit (...) {     debit(...) {
startStateMachine()    readBalance()      readBalance()
if (...)               if (checkA())      if (checkA())
  setState(2)             setState(3)       setState(5)
  debit()                 incBalance()      decBalance()
else                   else               else
  setState(1)            setState(4)        setState(6)
  credit ()              ISOException       ISOException
}                           .throwIt()         .throwIt()
                     }                  }
```

- Type confusion, executing an instruction on an object with a given type so this object is considered, in another code fragment, as another type.

The integrity of application data is often used in Java Card and is called secure storage. It mainly consists of a dual storage or a checksum in order to verify whether the modification of the field is done only through the VM. Another integrity check concerns the VM structure and in particular the frame context. In Bouffard, Lanet, and Iguchi-Cartigny (2011), the authors show how to modify the return address in the frame using unchecked local variable indices. Most of smart cards available on the web stores might be flooded by the modification of the CFG. Thus, it is possible to jump into an array which contains any shell code. It becomes obvious to dump the content of the EEPROM. A simple countermeasure consists in controlling the system data in the frame with a checksum. These data are the return address, the previous stack pointer and the context of the previous frame. A simple checksum of these data would mitigate the EMAN 2 attack.

In Bouffard, Lanet, and Iguchi-Cartigny, (2011), the possibility to modify the control execution of a byte code fragment has been demonstrated. The detection of such attack has been mainly studied in A. Séré's PhD thesis (Al Khary Séré A., 2010). He described several system countermeasures as the Field of Bits countermeasure, in Al Khary Séré, Lanet, and Iguchu-Cartigny in Evaluation of Countermeasures Against Fault Attacks on Smart Cards (2011), the Basic Block method and the Path Check method in Al Khary Séré, Lanet, and Iguchi-Cartigny in Checking the Paths to Identify Mutant Application on Embedded (2010):

- The Field of Bits consists in statically building a representation in an array associating the nature of the byte of the method: **1** representing an executable instruction, **0** a readable parameter. This information is sent to the VM, which is in charge of checking dynamically that each interpreted byte code is consistent with the associated bit;
- The Basic Block method generates statically the CFG of the method and at each end of a basic block, computes the value of the checksum. Dynamically, the VM is in charge of computing the value of checksum and checking the coherence with the pre-computed value at some predefined step: each entry point and each exit point;
- The Path Check method encodes statically the CFG as a field of bit and sends it to the VM with the application. Then the VM dynamically constructs its own field of bit according to the instructions executed. For each instruction, it becomes possible to verify if there is a divergence in the execution.

To prevent the execution of a shell code, there is the possibility to re-encode on the fly during the linking phase of the value of byte code. So if someone tries to execute an arbitrary array, he will not be able to obtain the desired behavior. Such a method is described in Razafindralambo, Bouffard, Thampi, and Lanet (2012), where the encoded value depends on a dynamic variable. They showed that using the jpc as a nonce is enough to avoid any brute force attack for guessing the scrambled value.

Several effects are possible for a type confusion using fault attacks:

- If the control flow is modified, the program counter can jump over several instruction leading to a storage of an operand of a given type into a local variable of incompatible type;
- As shown in Barbu, Thiebeauld, and Guerin (2010) a `checkcast` instruction can be bypassed by avoiding the dynamic control of a type cast;

- To desynchronize the control flow lead to a jump to an operand which will be executed as a byte code instruction. In turn this byte code instruction can lead to a type confusion.

There are a lot of possibilities to protect the data and the execution of a code into the VM. Unfortunately, if all of them are activated during the execution of an application, the performance of the smart card will drastically decrease reaching an unacceptable level. For that reason, such system based on countermeasures need to be activated only for some critical code sections and deactivated it once the code is no more critical. In the next section, we present a mechanism based on Java annotation to activate or deactivate these countermeasures.

2.6. The Annotation Mechanism

There are two ways to signal to a system in which the VM must enter/exit for the given mode. We can use annotations or a specific API (like `startTypedStack`(), `endTypedStack`(), *etc.*). The advantage of the API is that the loader of the VM does not need to be modified, while the advantage of annotation is the possibility to pre-compute statically information that will improve the run time check.

We made the choice of Java annotations which seems to be a more powerful mechanism for us. When the VM interpreter encounters an annotation, it switches to a "secure mode" and the scope of the annotation indicates the exit of this mode. The value provided within the annotation signifies the type of countermeasure which the developer needs for his application. The developers should keep in mind that activating a method in secure mode would imply that the parameters are correct. It is based on the paradigm of contract; if the calling context is correct, the VM guarantees an execution according to the value of the annotation parameter.

We have currently developed several annotations and we implemented them into our own VM. The first type of annotation concerns the integrity of the frame context. It consists in a new field that performs a *xor* with the three elements of the context frame (the previous stack pointer, return address and the context of the previous frame) avoiding any arbitrary modification of the return address. For the code integrity, we have developed several annotations in Al Khary Séré, Lanet, & Iguchi-Cartigny in Checking the Paths to Identify Mutant Application on Embedded (2010), but also recently the Java Card Linker (Hamadouche, et al., 2012) and the dynamic syntax interpretation (Razafindralambo, Bouffard, N Thampi, & Lanet, 2012) countermeasures have been added to our framework. The last one is a protection against illegal use of the Java stack. (Table 2)

The principle of the mechanism is divided into two parts: one part is off-card and the other part is executed on-card. Our module works on byte code, and it has sufficient computation power because all of the following transformations and computations are done on a server (off-card). It is a generalist approach which is not dependent on the type of application.

Table 2. List of annotations

Type	CTX_INTEGRITY	CODE_INTEGRITY	STACK_INTEGRITY
Parameters	CHECKSUM	FIELD_OF_BIT BASIC_BLOCK DYNAMIC_SYNTAX LINKER	TYPED_STACK

The fragment of code that follows displayed the use of an annotation on a debit method for a payment application. The `@SensitiveType` annotation (Listing 3) denotes that this method must be checked for integrity with the PATH-CHECK mechanism.

With this approach, we provide a tool that processes an annotated class file. The annotations become a Custom Component containing security information. This is possible because the Java Card specification (Sun, 2010) allows to add Custom Components to a class file. During the load-linking phase the VM process Custom Components if it knows how to use them. If does not, it ignores them. But in order to silently process the information contained in these Custom Components the VM must be modified. In the following section, we expose a complete example of a fault leading to type confusion.

2.7. Type Confusion

The following code is extracted from an attacked Java Card memory. The method ends by throwing the Java Card exception to PIN verification (code `0x6301`) and the jump at address `0x7404` throws this exception (*c.f.* Listing 4: Disassembling memory dump). If a fault is injected at this line, the transformed code will probably never throw the exception.

Listing 3. Sensitive type annotation example

```
@SensitiveType{
  sensitivity = SensitiveValue.CODE_INTEGRITY,
  proprietaryValue=”PATHCHECK”
}
private void debit(APDU apdu) {
   if (pin.isValidated()) {
     // make the debit operation
   } else {
      ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
   }
}
```

Listing 4. Disassembling memory dump

```
73F6: 18       aload_0
73F7: 7B 20 14 getstatic_a   0x2014
73FA: 8B 02 08 invokevirtual 0x0208
73FD: 32       sstore_3
73FE: 1A       aload_2
73FF: 03       sconst_0
7400: 1F       sload_3
7401: 8D 09 75 invokestatic  0x0975
7404: 60 2B    ifeq          0x2B
7406: 04       sconst_1
...
742F: 11 63 01 sspush        0x6301
7432: 8D 54 0D invokestatic  0x540D
7435: 7A       return
```

One can notice that after the execution of the instruction `ifeq`, the operand stack is empty. Now, consider that a laser hits the memory block that contains the `0x60` byte code, *i.e.*, `ifeq`, the resulting mutant is given in Listing 5: Mutant code.

After executing the `astore_0` instruction, the stack is empty and the mutant program is synchronized with the original program. A countermeasure based on the stack under/overflow will never detect the mutant. If a dynamic type verification had occurred, this mutant code should have been detected. In the original code the type system should evolve as described in Table 3. After executing the first instruction, a reference is pushed on the top of the stack. The second instruction pushes a value while the third consumes a reference and a value, and pushes a value after execution.

Now, examine the state of the stack with the mutant code describes in the Table 4. The instruction `ifeq` of the original code consumes a value and the `sconst_1` pushes a value. In the mutant code, the `ifeq` is replaced by a nop

Listing 5. Mutant code

```
...
7401: 8D 09 75 invokestatic  0x0975
7404: 00       nop
7405: 2B       astore_0
7406: 04       sconst_1
...
742F: 11 63 01 sspush        0x6301
7432: 8D 54 0D invokestatic  0x540D
7435: 7A       return
```

Table 3. Type evolution

Address	Code	Mnemonic	Stack After
73F6	18	aload_0	[ref]
73F7	7B 20 14	getstatic_a	[ref, val]
73FA	8B 02 08	invokevirtual	[val]
73FD	32	sstore_3	[]
73FE	1A	aload_2	[ref]
73FF	03	sconst_0	[ref, val]
7400	1F	sload 3	[ref, val, val]
7401	8D 09 75	invokestatic	[val]
7404	60 3B	ifeq 0x3B	[]
7406	04	sconst_1	[val]
…	…	…	…
742F	11 63 01	Sspush	
7432	8D 54 0D	invokestatic	
7435	7A	return	

Table 4. Type evolution of the mutant code

Address	Code	Mnemonic	Stack After
73FF	03	sconst_0	[ref, val]
7400	1F	sload 3	[ref, val, val]
7401	8D 09 75	invokestatic	[val]
7404	00	nop	[val]
7405	3B	pop	[]
7406	04	sconst_1	[val]
…	…	…	…
742F	11 63 01	sspush	
7432	8D 54 0D	invokestatic	
7435	7A	return	

which does not modify the state of the stack. The `astore_0` pops a reference from the stack, but cannot be executed because a value is on top of the stack. Obviously, it is easy to see how dynamic type verification increases the mutants' detection.

3. THE TYPE CLASSIFICATION

As we have seen, the most obvious countermeasures are related with under/overflow of the stack but their coverage is low; a lot of mutants can bypass these controls. The dynamic type verification is probably one of the most efficient countermeasures against mutant. It has to verify that the content on top of the stack is of the exact type expected by the next instruction. To obtain dynamic type verification, the VM needs to infer dynamically the type of locals and the type of each element on the top of the stack. But this is known to be costly in terms of computation and memory space because the VM must keep the stack evolution relating to type, which means to have a second stack where the type of each stack element must be stored. After executing an instruction, the VM must evaluate the type stack with regard to the executed instruction. Such a mechanism cannot be embedded into a resource constrained device like a smart card. Hereafter we propose a simpler mechanism for type classification based on implementing a pointer on the memory with no run time cost.

3.1. Principle

This countermeasure was presented in Dubreuil, Bouffard, Lanet, and Cartigny (2012). The cornerstone of our mechanism is to process references and values in a different way. It is possible to obtain a dynamic type checking by separating the operand stack into two areas one reserved for values and one for references. These two areas fill the same memory space used by the regular stack. The changes in our typed stack are just the place where you will find elements.

Here is an example showing how the typed stack works compared to a regular stack. If a program pushes on the stack one value and two references. To begin, it pushes a value, then pushes the first reference, and then finally the last reference is pushed (Table 5 through Table 7).

With the typed stack, there are two areas, one at the bottom for the values and the other one at the top for the references. The normal stack has one pointer called top of stack, but

Table 5. Typed stack 1

	Normal Stack	Typed Stack	
			⇓
⇑			⇑
	Value	Value	

Table 6. Typed stack 2

	Normal Stack	Typed Stack	
		Reference 1	
			⇓
⇑	Reference 1		⇑
	Value	Value	

Table 7. Typed stack 3

	Normal Stack	Typed Stack	
		Reference 1	
		Reference 2	⇓
	Reference 2		
⇑	Reference 1		⇑
	Value	Value	

for the typed stack we need two pointers, one pointing the top of the values and one for top of the references.

To reuse the example of a mutant application previously explained in the Table 4, at the nop instruction, address `0x7404`, the stack has only a value reference. The next instruction, `astore_0`, stores the last pushed value in the local 0. Here, we have two parts: the first part has a single-stack JCVM implementation and, the last pushed value is the return of the `invokestatic` instruction (address `0x7401`). This return type is a value. When the `astore_0` is executed, this return value is stored in a local variable whose type is a reference.

But on a typed stack JCVM implementation, when the `astore_0` is executed, the top part of the Java Card stack which contains the pushed reference value is empty. On the contrary, the bottom part contains the return of the `invokestatic` instruction. Thus, the JCVM detects an unexpected behavior. So this countermeasure prevents type confusion to be exploited as described in Iguchi-Cartigny and Lanet (2010).

The proposed countermeasure prevents type confusion attack and cover several attack paths, like `checkcast` described by Barbu, Thiebeauld, and Guerin (2010) or the Java Card stack overflow and underflow (Bouffard, Lanet, & Iguchi-Cartigny, 2011).

But this typed stack mechanism requires implementation of instructions in the VM to know which stack operand is used to get the elements. Most of Java instructions are typed, so it is easy to implement these instructions, knowing the type of elements, one instruction will push (pop) to (from) the stack. However, there are some untyped instructions and these instructions cause problems for the implementation of the VM. They cannot differentiate the references or values. These instructions are:

- `pop, pop2`
- `dup, dup2, dup_x`
- `swap_x`

The question is why these instructions are not typed. For example, a dup instruction which duplicates the last element stacked, the VM does not need to know whether it duplicates a value or reference with a simple stack. It must just go for the element pointed out by the top of stack. However, with a typed stack, the VM must know the type of the last element stacked to see whether it will get the element as a value or a reference. So with the typed stack, the VM cannot process these untyped instructions.

3.2. Modifying the Virtual Machine

Here we are modifying the VM by splitting the Java Card stack into two parts. In this new implementation of the VM we added a pointer to manage the values pushed on the stack. With two pointers, we can push (pop) the references to (from) the top of the stack and the values to (from) the bottom of the stack. In order to respect the backward compatibility of the JCVM, a Custom Component has been added to indicate if the VM runs the applet in typed stack mode or simple-stack mode.

Each of the untyped instruction must be removed to correctly run an application. Indeed, `pop` instruction, with dual-stack JCVM implementation, might be non-executable. We provide a way to protect your application against external modification (with a laser beam for example) to correctly run on a dual stack JCVM or run the applet in the JCVM without this countermeasure.

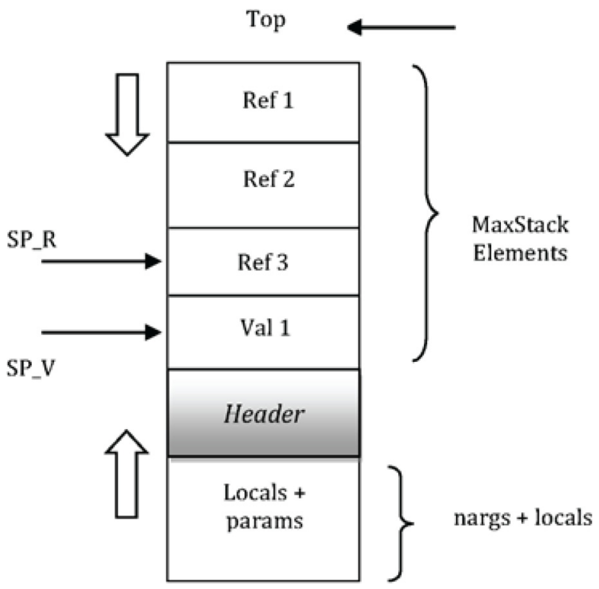
3.2.1. Virtual Machine Run Time

When the JCVM invokes a method, it pushes a frame from the top onto the Java stack. Then it loads the parameters, locals, header and the operands onto the stack. Compared to the single stack implementation, in dual stack approach the operand or element stack is split into two sections: the bottom area for values and top area for references. A value pointer *SP_V* points to the value and a reference pointer *SP_R* points to the reference of the current stack frame as shown in the Figure 2. In the dual stack implementation, the VM will identify the type of the data to be pushed or popped. If it is a value, it will be pushed (popped) to (from) the bottom of the operand stack and if it is a reference it is pushed (popped) to (from) the top of the operand stack. Subsequently, the value and reference pointers are updated according to the size of the value or reference and the operation performed. Once the method execution is finished, the stack frame is removed (popped) from the Java stack.

Dual stack is implemented in a protected mode when the JCVM interprets the annotation provided with the Custom Component. This mode can be enabled to execute sensitive methods. To prevent the type confusion, a pointer *SP_R* has been added to the stack. Dual stack operation is explained step by step in the algorithms.

Initialization of Java Card stack frame is explained in Algorithm 1. Once the stack frame is created, it is pushed onto the JCVM stack.

Figure 2. Representation of dual stack on JCVM



The stack pointers get updated accordingly. The reference pointer will change according to the size of the element and the `max_stack` value as shown in Algorithm 1: Push a Frame. Then the stack frame will load all the parameters, locals and header onto the stack and initialize all the stack frame pointers.

Operand stack management in dual stack is explained in Algorithm 2. When the execution of instruction starts, it will push (pop) value (reference) according to operations performed. Stack pointers *SP_V* and *SP_R* are updated based on executions.

Once the execution is finished, it will pop the current stack frame as shown in Algorithm 3.

Thereby using two pointers, type confusion is prevented. The switching between single and dual stack mode helps to manage the dual stack in secure mode.

3.3. Program Transformation

Although untyped instructions are rare in a Java Card program, we should be able to process these instructions properly. It requires transformation of the original program code so that the VM can run the program without errors. One solution is to replace untyped instructions by one or more other instructions which lead to the same result. These replacement instructions would use temporary variables to properly perform the treatment.

Since the method stack is local this transformation requires the analysis and modification of each method, one after the other. Before replacing untyped instructions we should have the stack history. With this information, the algorithm will be able to substitute untyped instructions. For example to replace a `pop`, knowing the type of the last element pushed on to the stack is enough; so if it is a reference just replace the `pop` by a `astore` into a local variable and if it is a value, replace by a `sstore` instruction. Analyzing the byte code instruction by instruction can help in extracting this data. Hence it is sufficient to perform a stack simulation alone as it is clearly known for each instruction what changes are made on the stack.

This byte code analysis is completely linear meaning that the instructions are read one after the other. However, jumps complicate the

Algorithm 1. Push a frame

```
/* Pointers declaration */
SP_V: value of the current stack frame
SP_R: reference of the current stack frame
 TOP: free space in the stack frame
 FP: first local of the stack frame

/* Initialization */
JCVM invokes a method
Push Frame /* to create a new frame for the
            * method */

/* Update Stack Pointers */
SP_R - =  m_localsize * SIZE_SHORT
        + m_maxstack;
SP_V + =  m_localsize * SIZE_SHORT;

/* Load methods parameters
 * and local variables and header */
```

Algorithm 2. Operand stack management

```
Until the method has instructions
  /* Push Value/ Reference */
  If opcode instruction push a value then
      /* Value is pushed to bottom of operand stack */
     SP_V = SP_V+SIZE_VALUE
  Endif
  If opcode instruction is push a reference then
     /* Reference is pushed to the top of the stack */
     SP_R = SP_R - SIZE_REFERENCE
  Endif

  /* Pop Value/Reference */
  If opcode instruction is pop a value then
     /* Value is removed from bottom of operand stack */
     SP_V = SP_V - SIZE_VALUE
  Endif
  If opcode instruction is pop a Reference then
     /* Reference is removed from the top of the stack */
     SP_R = SP_R + SIZE_REFERENCE
  Endif
End
```

Algorithm 3. Pop a frame

```
Reset Stack pointers
Reset Frame Pointers
Pop parameters and logical variables
```

analysis. The first approach is to go directly to the location pointed by a jump instruction and to continue the analysis. But it is not necessary to analyze the same instruction twice, and furthermore the analysis can even enter into an infinite loop. Therefore the analyzer stops when it finds that an instruction has not changed during the previous parse (fix point calculus).

Conditional jumps are another issue. If the condition is true, then the analysis must continue to the instruction pointed to the jump, and if it is false, the analysis must ignore the jump and continue. So the analysis must explore two branches and launch two sub-analyzers. Each of these analyzers must be run with an identical stack obtained just before the conditional jump.

In Algorithm 4, program transformation is used to perform the static analysis and the replacement of the untyped instructions.

During the load phase if the VM detects an untyped instruction and also the annotation requires the typed stack, the VM will return a security error code and refuses to load the application.

3.4. CapMap Integration

The CapMap (Razafindralambo, Bouffard, & Lanet, A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard, 2012) is a Java-framework which provides an easy way to parse and modify a CAP file. The CAP file is the file sent to the Java Card 2.x as a lightweight Java Class file.

This Java-library helps us to analyze the execution flow of the current Java Card applet. For each instruction, you can measure its impact on the stack (with the knowledge of the previously pushed type and value) in order to dynamically modify the CAP file. Then it is also possible to update each CAP file component to create a well-formed file. This tool is used to test card against logical attack.

Algorithm 4. Program transformation

```
/* Initialization */
If opcode is not already covered Then
   read opcode
   update stack history
   If opcode is a goto instruction
   Then go to the instruction pointed by goto
   If opcode is if instruction
     Then launch two analysis
/* One for the if statement and the other for the
 * else statement, both with the stack history
 * obtained with this if instruction.*/
    EndIf
    If opcode is untyped Then
       create replacement instructions from stack history
    EndIf
EndIf
```

In our case, the CapMap parses each CAP file to protect and for each applet method, verify if there are untyped operations on the stack. If there are some instruction blocks with untyped byte code, the CapMap modify these instructions as described in the Figure 3.

4. EXPERIMENTATION AND RESULTS

A countermeasure is affordable if its:

- latency (the number of instructions executed between the fault and the detection) is low;
- mutant detection success ratio is high;
- memory footprint is low.

The above three points are most important when designing a countermeasure for a smart card. The last point can be split into RAM and ROM usage knowing that the scarcest resource is the RAM. These metrics require the implementation of our methods in our own prototype while the latency and detection coverage can be obtained through a fault simulator (Machemie, Mazin, Lanet, & Cartigny, 2011).

Four Java Card applets have been used for the evaluation. Two applets are representative of the type of code that a MNO may want to add to their USIM Card. The first (Applet 1) is oriented geolocalization services, this applet is able to detect when the handset (the device in which the USIM card is inserted) is entering or leaving a dedicated or a list of geographical dedicated cells and then sends a notification to a dedicated service (registered and identified in the applet). The second (Applet 2) is more specialized to authentication services; the applet is able to provide a One Time Password (OTP) to the customer and/or an application in the handset. This OTP value is already shared and synchronized by the applet with a central server, which is able to check every collected OTP value by dedicated web services. The two other applications are oriented cryptography and scrambling operation (Applet 3 and Applet 4).

To replace an untyped instruction, the program transformer creates local variables which allow pushing or popping elements to (from) the stack, and it inserts new instructions to simulate the same effect than the untyped instruction. The metrics give us the occurrences of these instructions: `pop` (2%), `dup` (3%), `dup2` (<1%), and the others are extremely rare. As occurrences of these instructions are low in a Java Card application, there are not so many changes to do. If we want to remove one of these three instructions it does not cost much. It needs a new local variable to replace a `pop`; to replace a `dup`, needs to insert two instructions and a new local variable; and for a `dup2` instruction, insert five instructions and two variables. Moreover we could optimize local variables, taking those that are not used.

Figure 3. CapMap integration

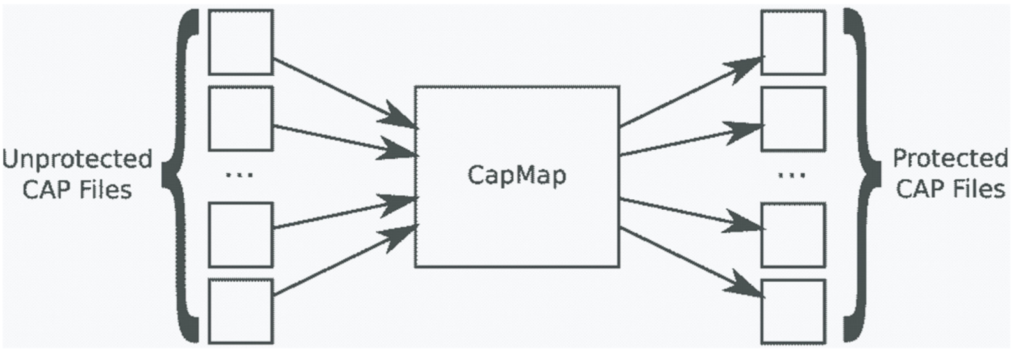


Table 8. Results

	Applet 1	Applet 2	Applet 3	Applet 4
Instructions count	4296	957	926	465
Untyped instructions	49 pop, 195 dup, 3 dup2	19 pop, 36 dup	4 pop, 17 dup, 4 dup2	10 pop, 6 dup
Instructions added	405	72	54	12
Locals added	27	16	7	5

Some statistics have been made on four applets to show the memory footprint when the untyped instructions are replaced by typed instructions (Table 8).

The `dup` instruction is the most commonly used untyped instruction after the `new` instruction. The applet size increases by 6% on an average. This growth is only due to `dup` instructions because `pop` replacement does not cost new instructions. And `dup2` instructions are very expensive but we can observe that they are not numerous. The number of local variables added is low and this number can be decreased if we optimize them. These metrics show that untyped instructions are rare, and the replacement of these instructions by typed instructions are affordable.

5. CONCLUSION

In this paper we presented a complete framework allowing the developers to activate or deactivate the system based countermeasures. A pre–processing phase to generate the Custom Component which can be interpreted by the VM had been proposed. Moreover we exposed a new approach to improve resistance of JCVM against type confusion attacks. As shown in the results (section 4), this proposed countermeasure is affordable and is fully backward compatible with the available platforms. This countermeasure needs an applet without untyped instructions on the stack. A static off-card tool is used for executing this operation. It could also provide a competitive advantage to a platform that implements this countermeasure. An application executed on a regular platform will be more prone to fault attack than the platform which is embedded with this countermeasure. This countermeasure is on a “secured mode” and this mode provides a protected environment to execute critical assets which are defined by the developer. We have seen here that the cost in terms of memory footprint was negligible while its detection capacity was better. Furthermore, the approach does not have any impact on the applicative development and the application transformer does not significantly increase the size of the application.

6. ACKNOWLEDGMENT

The work was partly funded by the French project INOSSEM (PIA-FSN2 - Technologie de la Sécurité et Résilience des Réseaux) and the Région Limousin.

7. REFERENCES

Al Khary Séré, A. (2010). *Tissage de contremesures pour machines virtuelles embarquées*. Unpublished PhD Thesis. France: Université de Limoges.

Al Khary Séré, A., Iguchi-Cartigny, J., & Lanet, J.-L. (2009, October 15th). Automatic detection of fault attack and countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security (WESS)*.

Al Khary Séré, A., Lanet, J.-L., & Iguchi-Cartigny, J. (2010, December). In T. Kim, Y. Lee, B. Kang, & D. Slezak (Eds.), Lecture Notes in Computer Science: Vol. 6485. Checking the paths to identify mutant application on embedded (pp. 459–468).

Al Khary Séré, A., Lanet, J.-L., & Iguchu-Cartigny, J. (2011, April). Evaluation of countermeasures against fault attacks on smart cards. *International Journal of Security and Its Applications*, *5*(2), 49–61.

Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., & Seifert, J. (2003). In B. Kaliski, Ç. Koç, & C. Paar (Eds.), Lecture Notes in Computer Science: Vol. 2523. *Fault attacks on RSA with CRT: Concrete results and practical countermeasures* (pp. 81–95). doi:10.1007/3-540-36400-5_20.

Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., & Whelan, C. (2006, February). The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, *94*(2), 370–382. doi:10.1109/JPROC.2005.862424.

Barbu, G., Thiebeauld, H., & Guerin, V. (2010). In D. Gollmann, J.-L. Lanet, & J. Iguchi-Cartigny (Eds.), Lecture Notes in Computer Science: Vol. 6035. *Attacks on Java Card 3.0 Combining Fault and Logical Attacks* (pp. 148–163). doi:10.1007/978-3-642-12510-2_11.

Blomer, J., Otto, M., & Seifert, J. (2003). A new CRT-RSA algorithm secure against Bellcore attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (pp. 311-320).

Bouffard, G., Lanet, J.-L., & Iguchi-Cartigny, J. (2011). In E. Prouff (Ed.), Lecture Notes in Computer Science: Vol. 7079. *Combined software and hardware attacks on the java card control flow* (pp. 283–296). doi:10.1007/978-3-642-27257-8_18.

Common Criteria Organization. (2012, September). *Common criteria for information technology security evaluation v3.1*. Retrieved from http://www.commoncriteria.org/

Dubreuil, J., Bouffard, G., Lanet, J.-L., & Cartigny, J. (2012, August 21st). Type classification against fault enabled mutant in java based smart card. In *Proceedings of the Sixth International Workshop on Secure Software Engineering (SecSE)* (pp. 551-556).

ESTI. (2005). *3GPP TS 31.102*. Technical Specification Group Core Network and Terminals.

Global Platform. (2011, January). *GlobalPlatform*. Retrieved from http://www.globalplatform.org/

Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., & Reygnaud, A. (2012, May 22sd). Subverting byte code linker service to characterize Java Card API. In *Proceedings of the Seventh Conference on Network and Information Systems Security (SAR-SSI)* (pp. 75-81).

Hemme, L. (2004). In M. Joye, & J.-J. Quisquater (Eds.), Lecture Notes in Computer Science: Vol. 3156. *A differential fault attack against early rounds of (Triple-) DES* (pp. 170–217). doi:10.1007/978-3-540-28632-5_19.

Iguchi-Cartigny, J., & Lanet, J.-L. (2010, November 1st). Developing a Trojan applets in a smart card. *Journal in Computer Virology*, *6*(4), 343–351. doi:10.1007/s11416-009-0135-3.

Ko, G. (2005). Fault attacks on Java card. Unpublished Masters Thesis, Eindhoven, Netherlands: University of Technology.

Machemie, J.-B., Mazin, C., Lanet, J.-L., & Cartigny, J. (2011, November 29th). SmartCM a smart card fault injection simulator. *IEEE International Workshop on Information Forensics and Security (WIFS)*, 1-6.

Noubissi, A., Al Khary Sere, A., Iguchi-Cartigny, J., Lanet, J.-L., Bouffard, G., & Boutet, J. (2009). Carte à puce: Attaques et Contremesures. *Majecstic*(1112). Retrieved from http://secinfo.msi.unilim.fr/software/cap-file-manipulator/

Piret, G., & Quisquater, J.-J. (2003). In C. Walter, Ç. Koç, & C. Paar (Eds.), Lecture Notes in Computer Science: Vol. 2779. *A differential fault attack technique against SPN structures, with application to the AES and Khazad* (pp. 77–88). doi:10.1007/978-3-540-45238-6_7.

Rankl, W., & Effing, W. (2000). *Smart card handbook* (2nd ed.). John Wiley & Sons.

Razafindralambo, T., Bouffard, G., & Lanet, J.-L. (2012). In N. Cuppens-Boulahia, F. Cuppens, & J. Garcia-Alfaro (Eds.), Lecture Notes in Computer Science: Vol. 7371. *A friendly framework for hidding fault enabled virus for Java based smartcard* (pp. 122–128). doi:10.1007/978-3-642-31540-4_10.

Razafindralambo, T., Bouffard, G., & Thampi, N. B., & Lanet, J.-L. (2012, October 11st). A dynamic syntax interpretation for Java based smart card to mitigate logical attacks. In *Proceedings of the International Conference on Security in Computer Networks and Distributed Systems (SNDS-2012)*.

Skorobogatov, S., & Anderson, R. (2003). In B. Kaliski, Ç. Koç, & C. Paar (Eds.), Lecture Notes in Computer Science: Vol. 2423. *Optical fault induction attacks* (pp. 31–48).

Sun. (2010). *Java card classic development kit 3.0 revenue release*. Retrieved from http://www.oracle.com/technetwork/java/javame/javacard/download/devkit/index.html

Sun. (2010). *Java card classic platform specification 3.0*. Retrieved from http://www.oracle.com/technetwork/java/javame/javacard

Wagner, D. (2004). Cryptanalysis of a provably secure CRT-RSA algorithm. In *Proceedings of the ACM conference on Computer and communications security* (pp. 92-97).

Ziegler, J., Curtis, H., Muhlfeld, H., Montrose, C., Chin, B., & Nicewicz, M. et al. (1996, January). IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, *40*(1), 3–18. doi:10.1147/rd.401.0003.

ENDNOTES

1 BSR: Bit Set or Reset

Jean Dubreuil finished his Bachelor's degree in Computer Science from the University of Limoges and started his Masters in Cryptology and IT-Security (CRYPTIS). He worked in SSD (Smart Secure Devices) team at XLIM as part of his internship on smart card software security. This work is the part of his thesis. His research interests include smart card security.

Guillaume Bouffard received his Master's degree in Cryptology and IT-Security (CRYPTIS) from the University of Limoges in 2010. He worked as a research engineer in SSD (Smart Secure Devices) team at XLIM labs for 6 months on smart card physical security before starting his PhD in 2011. His thesis is on the possibilities and issues of laser beam attacks on JCVM. His research interests include physical & logical attacks on embedded systems and smart cards.

Bhagyalekshmy N. Thampi received her engineering degree in Electronics and Communication from Anna University, India, and MSc. in Management of Embedded Electronic Systems from ESIGELEC, France. She is currently working as a Research engineer in SSD (Smart Secure Devices) team at XLIM, France. Her research interests include smart card security and EMC/EMI.

Jean-Louis Lanet is a Professor in the Computer Science Department at University of Limoges from 2007. He is also the team leader of SSD (Smart Secure Devices) research group at XLIM research lab. Prior to that, he was a senior researcher at Gemplus Research Labs (1996-2007). During this period he spent two years at INRIA (Sophia-Antipolis) (2003-2005) as an engineer at DirDRI (Direction des Relations Industrielles) and as a senior research associate in the Everest team. He started his career as a researcher at Elecma, Electronic division of the Snecma, now a part of the Safran group (1984-1995) and his field of research was on jet engine control. His research interests include security of small systems like smart cards and software engineering.