Security Automaton to Mitigate Laser-based Fault Attacks on Smart Cards

Guillaume Bouffard

XLIM UMR 7252 – University of Limoges,
123 Avenue Albert Thomas, 87060 Limoges CEDEX, France
E-mail: guillaume.bouffard@unilim.fr

Bhagyalekshmy N Thampi

University of Limoges,
123 Avenue Albert Thomas, 87060 Limoges CEDEX, France
E-mail: bhagyalekshmy.narayanan-thampi@xlim.fr

Jean-Louis Lanet

University of Limoges,
123 Avenue Albert Thomas, 87060 Limoges CEDEX, France
E-mail: jean-louis.lanet@unilim.fr

Abstract:

Security and attacks are two sides of the same coin in the smart card industry. Smart cards are prone to different types of attacks to gain access to the assets stored in it and that can cause security issues. It is necessary to identify and exploit these attacks and implement appropriate countermeasures to mitigate their effects. Fault attacks are one among them. They can introduce abnormal behaviour on the smart card environment. The redundancy is necessary to detect this change in their environment. In this work we propose an automatic method to obtain control flow redundancy using a security automaton to mitigate laser based fault attacks and hence implement a smart card countermeasure based on the combination of static analysis and dynamic monitoring method. This is a very cost effective approach which can identify and mitigate the effects of fault attacks in an efficient way.

Keywords: Smart Card; Laser Fault Attacks; Security Automata; Countermeasure

Biographical notes:

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 Smart Secure Devices (SSD) 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 Java Card virtual machine. His research interests include physical and logical attacks on embedded systems andsmart 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 worked as a

research engineer in Smart Secure Devices (SSD) team at XLIM, France. Her research interests include smart card security and EMC/EMI.
Jean-Louis Lanet is a Professor at the Computer Science Department, University of Limoges since 2007. 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 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, fault tolerant architecture and real time scheduling. Now his research interests include security of small systems like smart cards and software engineering.

1 Introduction

Smart card is an integrated chip with the smallest computing platform incorporating security at a system level. Smart cards find application in banking, SIM card, health insurance, electronic passports, *etc.* Data stored inside the card are extremely sensitive and requires to be protected from different types of hardware and software attacks. Attacks could be either purely logical which exploit software vulnerabilities or could be hardware type which abuse side channel vulnerability to access information about the protected security data/key or even cryptographic information. Logical attacks can be performed either by executing illegal instructions and/or accessing the secret information of a program. Hardware attacks can be realised using electromagnetic probes or laser beams. Among the hardware attacks, fault injection (FI) attacks using a laser beam is one of the most difficult to handle. It cause errors in the program execution, perturbations in the chip registers, bit flip, *etc.* which can be detected using some redundancies. Several investigations and approaches are proposed in various literatures and among them the use of security automaton and reference monitor are of larger interest. These techniques have emerged as a powerful and flexible method to enforce security policies over untrusted code.

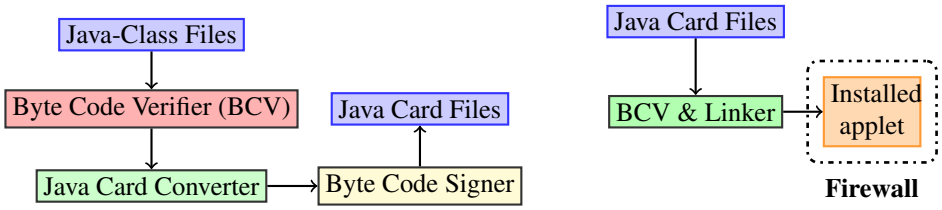
In Bouffard et al. (2013), we presented the general approach of using security automaton in a smart card. In this article, we detail how this approach has been implemented into a Java Card Virtual Machine (JCVM) and the obtained metrics. We evaluated the approach thanks to applets provided by our industrial partners, in term of memory footprint and execution time to check if this approach could be affordable in an industrial context.

This paper is organised as follows: section two describes the security architecture of a Java based smart card. Section three explains the perturbation attacks especially FI attacks on smart cards and their effects on program execution. The known fault detection mechanisms and their comparison are discussed in the fourth section. Section five presents our contribution and countermeasure. Final section gives the conclusions of our work.

2 Security Architecture of a Java-Based Smart Card

The Java Card platform is a multi-application environment, where the sensitive data of an applet shall be protected against malicious access from another applet or from the

external world. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers. In the smart card world, complying with the traditional enforcement process is not possible. The type verification is performed outside the card due to memory constraints. The Java Card platform provides further security enhancements, such as transaction atomicity, cryptographic classes and the applet firewall. The applet firewall replaces the class loader and security manager to enforce the sandbox security model. The Java Card security is ensured inside and outside the card due to the limited resources of the platform. During the conversion of the Java Class files, the semantics of the program is checked and signed (Figure 1) outside the card.



(a) *off-card* security model. (b) *on-card* security model.

Figure 1: Java Card Security Model.

For security reasons, the ability to download code into the card is controlled by a protocol defined by GlobalPlatform (2011). This protocol ensures that the owner of the code has the necessary authorisation to perform the action.

2.1 The Byte Code Verifier

Allowing code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (hand-made byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. Java is a strongly typed language where each variable and expression has a type determined during the compile-time, so that if a type mismatches arise from the source code, an error is thrown. The Java byte code is also a strongly typed one. Moreover, local and stack variables of the virtual machine have fixed types even in the scope of a method execution. None of the type mismatches are detected during the run time which can allow the malicious applets to exploit this issue. For example, pointers are not supported by the Java programming language although they are extensively used by the Java Virtual Machine (JVM) where object references from the source code are relative to a pointer. Thus the absence of pointers reduces the number of programming errors. But it does not stop attempting to break security protections with unfair uses of pointers.

The Byte Code Verifier (BCV) is an essential security component in the Java sandbox model: any bug created by an ill-typed applet could induce a security flaw. The byte code verification is a complex process involving an elaborate program analysis using a very costly algorithm in terms of time consumption and memory usage. For these reasons, many

cards do not implement this kind of component and also it relies on the responsibility of the organisation which provides signature to ensure the code of the applet is well-typed.

2.2 The Java Card Firewall

The separation of different applets is enforced by a firewall which is based on the package structure of Java Card and the notion of the contexts. When an applet is created, the Java Card Runtime Environment (JCRE) uses a unique Applet IDentifier (AID) to link it with the package where it has been defined. If two applets are an instance of classes of the same Java Card package, they are considered to be in the same context. There is also a super user context called JCRE. Applets associated with this context can access the objects from any other contexts on the card.

Each object is assigned to a unique owner context, which is the context of the created applet. An object's method is executed in the context of the instance. This context provides information which will or will not allow access to another object. The firewall prevents a method executing in one context from accessing any attribute or method of objects to another context.

There are two ways to bypass a firewall. One is through the JCRE entry points and the other one is by shareable objects. JCRE's entry points are the objects owned by JCRE, specifically entitled as objects that can be accessed from any context. A significant example is an APDU buffer which contains the sent and received commands from the card. This object is managed by JCRE and in order to allow applets to access this object, it is designated as an entry point. Another example is the elements of the table containing the AIDs of the installed applets. Entry points can be marked as temporary. References to temporary entry points cannot be stored in objects and this rule is enforced by the firewall.

2.3 Execution Consistency

By nature, smart card involves in a hostile environment. Due to the fact that its power, clock and reset are provided by the external world, the card must be protected against any modification of these parameters. Software processes often rely on internal data consistency, and can have an erratic behaviour in case of power disruption.

Java Card introduces a transaction mechanism that guarantees atomicity. It makes sure that all the operations within a transaction is completed. At the end of each transaction, a commit command confirms the completion of the previous operations. If the transaction is aborted by the program or due to power shortage, the mechanism confirms that all the earlier operations within a transaction have set back to their previous state. In this way, it is possible to maintain the internal consistency of the related data.

2.4 The Sharing Mechanism

To support cooperative applications on one-card, the Java Card technology provides well-defined sharing mechanisms. The Shareable Interface Object (SIO) mechanism is a system in the Java Card platform meant for the collaboration of the applets. The `javacard.framework` package provides a tagging interface called `Shareable` interface and the methods described in the the `Shareable` interface are available through

the firewall. Any server applet which provides services to other applets within the Java Card should define the exportable services in an interface tagged as shareable.

2.5 The CAP File

The CAP (*Converted APplet*) file format is based on the notion of components that contain specific information from the Java Card package. It is specified by Oracle (2011) which consists of eleven standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool`, `Reference Location` and `Descriptor`. The `Debug` component is only used for the debug process. Moreover, the targeted JCVM may support user's `custom` components.

2.6 Synthesis

Smart card security is a complex problem with different perspectives, however, the products based on JCVM have passed the real-world security evaluations successfully for major industries around the world. Java Card is also a platform that has cleared high level security evaluations for issuance by banking associations and by leading government authorities. It has also achieved compliance with FIPS 140-1 certification scheme. Still, implementations have undergone several attacks, particularly perturbation attacks.

3 Perturbation Attacks on Smart Cards

In general, a fault is an event that changes the behaviour 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, 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 analyse the errors generated by these faults more precisely.

3.1 Fault Attacks

Smart card is a portable device for which a smart card reader provides external power and clock sources to operate. The reader can be replaced with a specific equipment to perform the attacks. With short variations in the power supply, it is possible to induce errors on smart card's internal operations. These perturbations are called spike attacks, which may induce errors in the program execution. Latter aims at confusing the program counter and it can cause the improper working of conditional checks, a decrease in loop counters and the execution of arbitrary instructions. A reader like Microprocess MP300 can be used to provide a glitch attack. As described by Anderson & Kuhn (1997), Boneh et al. (1997), Joye et al. (1997), a glitch incorporates short deviations beyond the required tolerance from a standard signal bounds. It can be defined by a range of different parameters and can be used to inject memory faults as a faulty execution behaviour. Hence, the possible effects are the same as in spike attacks.

An idea to inject physical faults to shift the semantics of an application has been emerged recently. Based on the FI, an attacker can modify a part of the memory contents

or a signal on an internal bus since from an applet's execution stage, which can lead to an exploitable deviant behaviour. So the application mutates and executes a malicious byte code that can break the security model. The fault attacks are used to attack cryptographic algorithm implementations as presented by Aumüller et al. (2002), Hemme (2004), Piret & Quisquater (2003).

Barbu et al. (2010) proposed a way to bypass the embedded smart card BCV. To accomplish that, a correct applet was installed which contains an unauthorised cast between two different objects. Statically, this applet is in compliance with the Java Card security rules. If a laser beam hits the bus in such a way that the cast type check instruction is not executed, this applet becomes a malware. Moreover, the authors were able to load applications into the targeted Java Card. The authors implemented three Java classes defined in the Listing 1. The first one is the class `A` which contains 255 byte fields type. The second one is the class `B` that has a short-integer field type and the last one is the class `C`, referred to an instance of `A`.

Listing 1: Classes used to create a type confusion.

```
public class A {
  byte b00, ..., bFF;
}
```

```
public class B {
  short addr;
}
```

```
public class C {
  A a;
}
```

A `checkcast` verification is done in (line 9) for the applet with the code shown in the Listing 2. This applet becomes a malware because a laser beam hits the bus in such a way that the `checkcast` instruction is temporally avoided. With this invalid cast, the authors succeeded to obtain a window which allows to access to the content of the smart card memories.

Listing 2: `CheckCastApplet` class.

```
1  public class CheckCastApplet extends Applet {
2      B b; C c;
3      ... // Constructor, install method, ...
4      public void process(APDU apdu) {
5        ...
6        switch (buffer[ISO7816.OFFSET_INS]) {
7          case INS_ILLEGAL_CAST:
8            try {
9              c = (C) ( (Object) b ); // Checkcast check
10             return; // Success, return no error status word
11           } catch (ClassCastException e) {
12             /* Invalid cast is detected */
13           }
14           ... // more later defined instructions
15 }   } }
```

An approach to disturb the Control Flow Graph (CFG) of an applet by injecting laser beam into the non-volatile memory of a smart card was proposed by Bouffard et al. (2011). This attack was performed on a `for` loop as described in the Listing 3. The byte code version of this loop is presented in the Listing 4. This attack can also be extended with other type of loop or condition.

Listing 3: A `for` loop sample.

```
for (short i=0 ;
     i<n ; ++i){
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
...
 // Few instructions
 // are hidden here
 // for a better
 // understanding
...
 bar = foo; foo = bar;}
```

Listing 4: Associated byte codes of the loop listed in the Listing 3.

```
sconst_0
sstore_1
sload_1
sconst_1
if_scmpge_w       00 7C
aload_0
bspush            BA
putfield_b        0
aload_0
getfield_b_this   0
putfield_b        1
// Few instructions
// are hidden here
// for a better
// understanding
aload_0
getfield_b_this   1
putfield_b        0
sinc              1 1
goto_w            FF17
```

The Java Card specification defines two instructions to rebranch a loop, a `goto` and the `goto_w` . The first one branches with a 1-byte offset and the second one takes 2-byte offset. Since the smart card's memory manager stores array data after the memory byte code, a laser fault on the high part of the `goto_w` parameter can shift the backward jump to a forward one and the authors succeeded to execute the contents of an array. However, the knowledge on Java Card's internal reference is needed to execute a rich shellcode. Hamadouche et al. (2012) described a way to obtain Java Card's API addresses embedded in the card. With this attack, it is possible to know the internal references of the Java Card.

Lancia (2012) exploited the Java Card instance allocator of JCRE based on high precision FI. Each instance created by the JCRE is allocated in a persistent memory. The Java Card specification Oracle (2011) provides some functions to create transient objects. The data of the transient object are stored in the RAM memory, but the header of this object is always stored in the persistent memory. On the modern Java Card using Memory Management Unit (MMU), references are represented by an indirect memory address. This address is an index to a memory address pool which in turn refers to a global instance pool managed by the virtual machine (Figure 2).

3.2 Fault Models

As shown by Bouffard et al. (2011), it is possible to induce a laser beam into the memory cells since the silicon layer of smart card chip is visible. These memory cells are found to be sensitive to light. Due to photoelectric effect, modern lasers can be focused on relatively small regions of a chip and dynamically modify the execution flow as explained by Barbu (2012).

It is necessary to know the effects of a fault attack on smart cards to detect it. Fault models have been already discussed in details by Blömer et al. (2003), Wagner (2004).

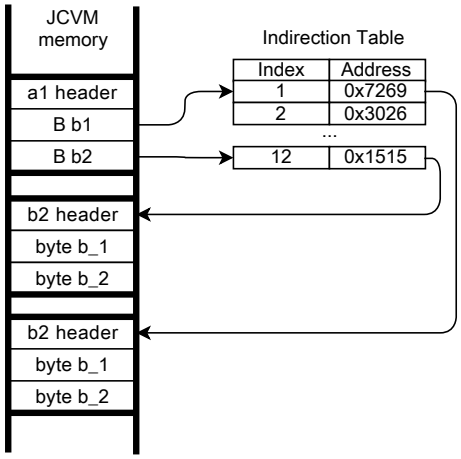


Figure 2: Global Java Card Instance Pool Mechanism.

Using the precise bit error model, an attack was described by Skorobogatov & Anderson (2002). But it is not realistic on current smart cards since the modern components implement hardware security mechanisms, like error detection and correction code or memory encryption. During the program execution, an attacker physically injects energy into a memory cell to change its state. Thus, up to the underlying technology, the memory physically takes the value `0x00` or `0xFF`. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address, and an encryption key). To be as close as possible to the reality, we chose the most realistic fault model, the precise byte error. So an attacker:

- can make FI at a precise clock cycle (can target any operation he wants),
- can only set or reset a byte to `0x00` or to `0xFF` up to the underlying technology (bit set or reset fault type), or he can change this byte to a random value beyond his control (random fault type),
- can target any of the memory cell he wants (can target a specific variable or register).

Nowadays, the Information Technology Security Evaluation Facilities (ITSEF) are using low power laser diodes to illuminate the smart card. This technology drastically reduces the charging period of the laser. Taking this approach as a hypothesis, the attacker can now attack a program and a given countermeasure at the same time which makes the traditional applicative countermeasures ineffective.

3.3 Effects of the Fault Attacks on the Program Execution

In this work, only a single fault is considered. However our proposed mechanism supports dual faults since it 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 the one executed by

the card. The data of a program are also a sensitive asset to be protected. With a single fault, an attacker can permanently or temporarily, modify a sensitive information. In particular, it can affect the variables used in any evaluation instruction like never start a loop, ignore initialisation 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.

As seen, one of the effects of the fault is to modify the value of a register. The JVM registers are highly sensitive. For example, the Java Program Counter (JPC) can be altered by a fault. A fetch sequence of the byte code to be interpreted is shown in Listing 5. In this interpreter loop, the address of the function corresponding to the byte code to be interpreted is stored into the array `bytecode_table`. The index is obtained through `vm_pc` which is pointed to the content of the currently executed method.

Listing 5: Fetch of the next instruction.

```
558  handler = bytecode_table[*vm_pc];
559  vm_pc++;   // jpc is updated
560  bc_action = handler();
```

This JCVM was compiled for an ARM7 target and a code fragment is given in Listing 6. In line 1809, the local variable that stores the `vm_pc` is loaded into `r3` which corresponds to the 32 bit instruction `012083E2` which is in fact `E2 83 20 01`, regardless of the endianess representation. Thus, if a laser hits and nullifies the third byte, the instruction becomes `E2 83 00 01` which corresponds to the instruction `add r0, r3, #1` storing into `r0` the new value of the `vm_pc` variable. But the real storage is line 1811 and it stores the content of the `r2` register which has the value stored line in 1802.

Listing 6: Fetch at the binary level.

```
     .loc 1 558 0 ; j_vm.c:558 handle= bytecode_table[*vm_pc];
     80319FE5        LDR     r3, .L104+24
1800 003093E5        LDR     r3, [r3, #0]
     0030D3E5        LDRB    r3, [r3, #0]
     0320A0E1        MOV     r2, r3
     74319FE5        LDR     r3,.L104+28
     023193E7        LDR     r3,[r3, r2, asl #2]
1805 18300BE5        STR     r3,[fp, #-24]
     .loc 1 559 0 ; j_vm.c:559    vm_pc++;
     64319FE5        LDR     r3, .L104+24
     003093E5        LDR     r3, [r3, #0]
     012083E2        ADD     r2, r3, #1
1810 58319FE5        LDR     r3, .L104+24
     002083E5        STR     r2, [r3, #0]
     .loc 1 560 0 ;  j_vm.c:560   bc_action = handler();
     18301BE5        LDR     r3, [fp, #-24]
     0FE0A0E1        MOV     lr, pc
1815 13FF2FE1        BX      r3
     0030A0E1        MOV     r3, r0
```

Within the code fragment shown above in the Listing 6, one can see that a simple fault can lead to the interpretation that jumps are not to the next expected byte code. Especially it can avoid a given method invocation, ignore a condition loop or it can jump to a specific statement. Likewise, by modifying the destination or source register, an attacker can modify the value returned by a function which allows the execution of a sensitive code without

authorisation, avoiding initialisation of variables. He can also generate a faulty condition to jump. If the destination of the jump corresponds to an operand instead of a byte code, he can execute a different program often called mutant program in the literature.

Listing 7: Faulty fetch of the next instruction.

```
558  handler = bytecode_table[*vm_pc];
559  vm_pc= * vm_pc ;
560  bc_action = handler();
```

Now, the binary code has a new semantics after the fault occurs which is showed in Listing 7. As one can notice, the `vm_pc` gets the value of the current byte code to which it points, that can lead to a jump anywhere, especially into a static array stored just after the method.

Evaluating the effects of fault on a binary program is quite impossible due to the combinatorial possibilities. An analysis is often dedicated to a given target and a small function. Up to now, only generic solutions are applied and are often at the applicative level. Checking the integrity of the code with some hash functions is useless against transient faults, since the checked code is not the one executed by the system.

4 Fault Detection Mechanisms

The fault detection mechanism can be classified into three countermeasure approaches: static, dynamic and mixed.

4.1 Static Countermeasure Approach

Static countermeasures ensure whether each test is done correctly and/or the program CFG remains unchanged as described by the developer. It is done at the applicative layer. Here the main advantage is that the developer has the knowledge of the assets to be protected. Apart from that, the knowledge of fault attacks is also very important to implement security features. Two examples of applicative countermeasures are explained below.

The redundancy `if-then-else` statement can be used to improve the security of the branching statement to verify if a test (*i.e.* a sensitive condition `if`) is performed correctly. For example, in order to verify a PIN code, a call to `pinIsValidated()` should be performed, which returns `true` if the PIN code has been verified previously. `pinIsValidated()` is provided by the PIN Java-interface. If the PIN code is not validated, the program will check it again whether the condition did not occur before executing an operation. If the condition occurred without having a call to the adequate method (*i.e.* the `verifyPIN()`) that means some external phenomenon has modified the state of the PIN object during the transfer of data on the bus.

Indeed, if a fault is injected during an `if` condition, an attacker can execute a specific statement without a check. In real time, a 2^{nd} order FI is difficult with a short delay between two injections. A 2^{nd} order `if` statement can be used to verify the requirements needed to access a critical operation in order to prevent a faulty execution of an *if-then-else* statement. An example of this kind of implementation is listed in the Listing 8. The problem with a

Listing 8: Protected `if` statement.

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

secure `if` condition is that the CFG of the program is not guaranteed.

The second applicative countermeasure is a step counter approach. The developer can implement this method as described in the Listing 9 to make sure that the control flow has been respected and also the correctness of the program execution flow. Here several check points can be inserted and each node of the CFG defined by the developer, is verified during the runtime. If a step counter is set with a wrong value at the execution time, a faulty behaviour can be detected. In the Listing 9, a variable `step_counter` is initialised and incremented until it reaches a sensitive node. At that particular point, its value is compared with the expected value and the if a mismatch arises, it can cause an unexpected behaviour, and a security action must be taken.

Listing 9: Step counter.

```
short step_counter=4;
if(step_counter==4) {
  // Critical operation 1
  step_counter++;
} else {/*Attack detected!*/}
/* ... */
if(step_counter==5) {
  // Critical operation 2
  step_counter++;
}else{/*Attack detected!*/}
```

4.2 System based or Dynamic countermeasure approach

In the applicative countermeasure approaches, the developer himself is in charge of securing his code. Another approach is a system based countermeasure, which is used to provide security mechanisms by the system itself. Most of these countermeasures need an automatic off card static analysis by the applet in order to reduce the run time cost.

To prevent the modification of the dynamic elements (stack, data, *etc.*), and also to ensure integrity, the smart cards can implement countermeasures on stack and data. 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. (2013). Their countermeasure implements the principle of a dual stack where each value is pushed from the bottom and growing up into the stack element. 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. The memory can be encrypted to ensure the confidentiality of the code. For using a more affordable countermeasure, Barbu (2012) purposed a method to scramble the code. Unfortunately, a brute force attack can bypass a scrambled memory. Razafindralambo et al. (2012) improved this countermeasure based on a randomised scrambling operation to protect the code confidentiality.

Enabling all the countermeasures during the complete program execution is too expensive for the card to afford and also it is not required. Hence to reduce the implementation cost of the countermeasure, Barbu et al. (2012) proposed user-enabled countermeasure(s) in which the developer has the choice to enable a specific countermeasure for a particular code fragment.

Recently, Farissi et al. (2013) presented an approach based on artificial intelligence, particularly in neural networks. This mechanism is included in the JCVM. After a learning step, this mechanism can dynamically detect the abnormal behaviour of each smart card's program.

4.3 Mixed Countermeasure Approach

Unlike the previous approaches, mixed methods use off-card operations where some computations are performed for embedded run-time checks. This way offers a low cost with respect to the costly operations realised outside the card.

To ensure the code integrity, Prevost & Sachdeva (2006) patented a method, in which a hash value is computed for each basic block of a program. The program is sent to the card with the hash of each basic block. During the execution, the smart card verifies this value for each executed basic block and if a hashsum is wrong, an abnormal behaviour is detected.

Al Khary Séré (2010) 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, which is to be provided to the smart card through a custom component. The smart card checks the correctness of the current CFG dynamically. Since there are off-card operations, this countermeasure has a low footprint in the smart card's runtime environment.

In this section, we described some published countermeasures to prevent FI attacks. A summarize of the assets protected by each countermeasure is shown in the Table 1.

5 Security Automata and Reference Monitor

Detecting a deviant behaviour is considered as a safety property, *i.e.* properties that state "*nothing bad happens*". A safety property can be characterised by a set of disallowed finite execution based on regular expressions. The authorised execution flow is a particular safety

Countermeasures	Code Protection		Data Protection	
	Integrity	Confidentiality	Integrity	Confidentiality
`if` statement	✓			
Step counter	✓			
Checksum			✓	
Dubreuil et al. (2013)			✓	
Barbu (2012)		✓		✓
Farissi et al. (2013)	✓			
Prevost & Sachdeva (2006)	✓			
Al Khary Séré (2010)	✓			

Table 1 Sum up of the FI protection mechanisms.

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 authorised behaviours. The key point is that this property can be encoded by a finite state automaton, while the language recognised will be the set of all authorised partial traces of events.

5.1 Principle

Schneider (2000) 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 S is a set of input symbols, *i.e.* the set of security relevant actions. The security automaton processes a sequence of input symbols s_1, s_2, …, s_n 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 here is a redundancy of the control flow. In the first approach, the automaton that verifies the control flow could be inferred using an inter procedural 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 start either by the 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 authorised 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 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 actually consider the double FI possibility for an attacker. If needed, it is possible to protect the automaton with an integrity check verification before each access into the automaton.

5.2 Security Automaton Included in a JCVM

We present here a code fragment 10 extracted by Girard et al. (2010) from the Internet protocol payment defined by Gemalto. It starts by an array initialisation with a loop followed by a call to the method `update()` in order to initialise the PIN code and a call to `register()` to register the applet into the card.

Listing 10: Source code of the payment applet.

```
protected Protocolpayment (byte[] buffer, short offset, byte length) {
   A[0] = 0;  // init. of array A
   for (byte j = 0; j < buffer[(byte)(offset+12)]; j++)
      D[j] = 0; // init. of array D
   pin = new OwnerPIN((byte) TRY_LIMIT, (byte) MAX_PIN_SIZE);
   pin.update(myPin, (short) START_OFFSET, (byte) myPin.length); // initialisation of pin
   register();  // register this instance
}
```

The set S is made of elements of a language which expresses the control flow integrity policy, *i.e.* all the binary instructions controlling the program flow : `ifeq`, `ifne`, `goto`, `invoke`, `return`, ... plus the dummy instruction `join`. In this example, the number of loop iterations cannot be statically computed, but it can be represented by a regular expression. The CFG of this program is given in Figure 3.

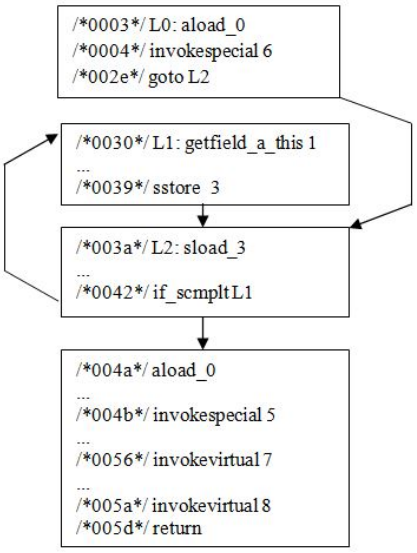


Figure 3: CFG of the applet constructor.

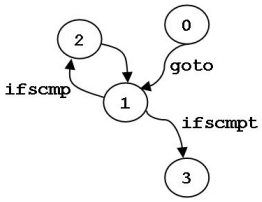


Figure 4: Applet constructor automata.

The first block ends with a `goto`, the end of the second block precedes a label `join` and the last one finishes with `return`. Inside basic block, they are calls to other methods, the first one is the constructor of the super class. In the fourth block we have a call to the constructor of `OwnerPIN` followed by the method `update` and finally the `register`.

Each invoked method has its own CFG and its own automaton. This automaton represents an abstraction of the program (its CFG) and is used by the monitor to control the execution. The automaton can be built statically off card and loaded with the applet as an optional component of the CAP file or the construction of the automaton can be done by the card itself while loading the code. The code is always loaded in a safe environment and there should not be any attack during this phase. A simple integrity check will preserve the code or the automaton to be altered before being stored into the card and this point is discussed later.

The trace recognized for this method would be : (`goto`, `ifscmpt`*, `join`, `return`). The automaton that recognizes this trace is shown in Figure 4. The condition of the loop can be evaluated at least once. In fact, the trace can be more precise: the call to the methods and use of reference to checks can be taken into account, if the control flow has been correctly transferred to the called method. Thus, the recognized trace becomes: (`invokespecial 6`, `goto`, `ifscmpt`*, `join`, `invokespecial 5`, `invokevirtual 7`, `invokevirtual 8`, `return`), as given in Figure 5.

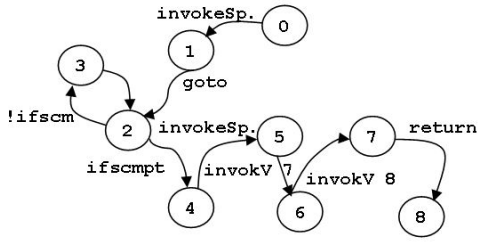


Figure 5: Applet constructor automata.

Such a state machine can be easily represented by an array (see Table 2), allowing the system to check if the current state allows to change the state to a requested one for each

Table 2 Basic representation of the automata.

δ \ q	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
`invokespecial 6`	q_1							
`goto`		q_2						
`join`				q_2				
`ifscmpt`			$q_{3,4}$					
`invokespecial 5`					q_5			
`invokevirtual 7`						q_6		
`invokevirtual 8`							q_7	
`return`								+

transition function. Moreover, keeping trace of the JPC allows a fine grain control of the CFG. For example, if the JCVM encounters the instruction `goto label`, it checks if in the current state (says for example q_1) such an instruction is allowed, if not, it takes an adequate countermeasure. If in current state the instruction is allowed, the JCVM checks whether the destination is an expected one, *i.e.* q_2 by verifying the label of the instruction or the token of the invoked method. If the instruction is a `return`, it verifies either it is the last instruction or the next instruction has a label.

5.3 The Reference Monitor

The control of the transition functions is quite obvious. Once the automaton array has been built statically either *off-card* or during the linking process, each Java frame is updated 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 security automaton method in its own Java frame for each method. Knowing the current state and the current instruction, it is easy to check the source and the destination while executing an the instruction related to control flow. Unfortunately such a matrix is not compatible with a highly constrained device like the smart card. Thus, we need to have a compact representation inside the card.

Listing 11: 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 -=2;
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 11, the test (in line 2) verifies that the currently executed instruction 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 whether 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 12: Transition function for the `ifle` byte code (target jump).

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

In the Listing 12, the last part of the `ifle` byte code also checks the destination JPC matches with the next state and then updates the current state.

6 Metrics

The modification of the JCVM affects the interpreter and potentially the linker-loader if one prefers to build an on-the-fly state machine instead of implementing it as an additional component of the CAP file. In this proof of concept, we implement it as an additional component. The overhead must be evaluated in terms of ROM, RAM and EEPROM memory. The RAM being the more scarce resource, an optimisation is needed for the implementation with this criteria. The Java frame has been modified by adding a byte for storing the current value of the state. The cost of the RAM overhead is one byte per method call. The second memory to be optimised is the EEPROM. It contains the matrix storing the automaton `SM` for each method. It can be written once during the load and read until the applet is removed from the card. It is a two dimensional array with a particular entry to manage instructions having multiple jumps like `tableswitch`, `lookupswitch`, …We did not make an optimisation of this structure in order to maintain a direct access in $\mathcal{O}(1)$. For an already installed Java Card application (API, Romised Applets, *etc.*) this table is burned in the ROM area which is less constrained. So the memory overhead is minimalist for the RAM, and for the EEPROM, it depends on the structure of methods for the application uploaded in *post-issuance*.

The second metrics is about the execution time overhead. Each Java Card instruction requires two cycles: *prefetch* and *execute*. The *prefetch* is fixed regardless of the automaton implementation. In our implementation of the Java Card on an ARM7, it costs $0.96\,\mu s$. The *execute* cycle costs, for the `if_scmplt`, $0.615\,\mu s$. In fact the modification of the interpreter increases the execution time by $0.332\,\mu s$. The instruction that needed $1.575\,\mu s$ requires now $1.907\,\mu s$ says an overhead of 19%. But only the instructions that change the control flow are modified, *i.e.* 45 instructions over the 184 instructions of the Java Card set. Of course the overhead depends on the used instructions in the method. In the given example, Listing 10, only 7 instructions over the 93 have an overhead.

7 Related Works

Aktug (2008) 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 1st order logic annotations. They use a weakest precondition computation that works as same as the annotation propagation algorithm that is used by Pavlova et al. (2004) to produce a fully annotated, verifiable program for the Java Card. This allows the use of Java Modeling Language (JML) verification tools, to verify the actual policy adherence. Such a static approach cannot be adopted here due to the dynamic nature of the attack.

The only application of the security automaton for smart card was presented by McDougall et al. (2004) where the concept of policy automaton which combines the defeasible logic with a state machine was used. It represents the complex policies as a combination of the basic policies. A tool has been implemented for performing policy automaton analysis and checking policy conflicts. A code generator was used to implement the transition functions that creates a Java Card applet. It was concerned mainly to enforce invariants in the application.

8 Conclusion

In this work we introduced and implemented a countermeasure to detect the FI attacks for smart cards. We presented an automatic method to obtain control flow redundancy using a security automaton executed in the kernel mode. The automaton was generated automatically during the linking process or by an off-card process. This automaton is modeled by a regular expression which describes each instruction to be executed. We also presented the metrics of our Java Card implementation on an ARM7 processor. From the implementation we concluded that the proposed method is a cost effective and efficient one.

This technique is not only limited to CFG properties but it can be used for more general security policies expressed as *safety properties*. It is interesting to check whether some security commands have already realised before executing a sensitive operation. Some are memorised in a 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 here is to find a right trade-off between the highly secured system with a poor run-time performance and an efficient system with less security.

References

Aktug, I. (2008), Algorithmic Verification Techniques for Mobile Code, PhD thesis, KTH, Theoretical Computer Science, TCS. QC 20100628.

Al Khary Séré, A. (2010), Tissage de contremesures pour machines virtuelles embarquées, PhD thesis, University of Limoges.

Anderson, R. J. & Kuhn, M. G. (1997), Low Cost Attacks on Tamper Resistant Devices, *in* B. Christianson, B. Crispo, T. M. A. Lomas & M. Roe, eds, 'Security Protocols Workshop', Vol. 1361 of *Lecture Notes in Computer Science*, Springer, pp. 125–136.

Aumüller, C., Bier, P., Fischer, W., Hofreiter, P. & Seifert, J.-P. (2002), Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures, *in* Jr. et al. (2003), pp. 260–275.

Barbu, G. (2012), On the security of Java Card platforms against hardware attacks., PhD thesis, Grant-funded PhD with Oberthur Technologies and Télécom ParisTech.

Barbu, G., Andouard, P. & Giraud, C. (2012), Dynamic Fault Injection Countermeasure - A New Conception of Java Card Security, *in* Mangard (2013), pp. 16–30.

Barbu, G., Thiebeauld, H. & Guerin, V. (2010), Attacks on Java Card 3.0 Combining Fault and Logical Attacks, *in* D. Gollmann, J.-L. Lanet & J. Iguchi-Cartigny, eds, 'CARDIS 2010', Vol. 6035 of *Lecture Notes in Computer Science*, Springer, pp. 148–163.

Blömer, J., Otto, M. & Seifert, J.-P. (2003), A new CRT-RSA algorithm secure against bellcore attacks, *in* S. Jajodia, V. Atluri & T. Jaeger, eds, 'ACM Conference on Computer and Communications Security', ACM, pp. 311–320.

Boneh, D., DeMillo, R. A. & Lipton, R. J. (1997), On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract), *in* W. Fumy, ed., 'EUROCRYPT', Vol. 1233 of *Lecture Notes in Computer Science*, Springer, pp. 37–51.

Bouffard, G., Iguchi-Cartigny, J. & Lanet, J.-L. (2011), Combined Software and Hardware Attacks on the Java Card Control Flow, *in* E. Prouff, ed., 'CARDIS 2011', Vol. 7079 of *Lecture Notes in Computer Science*, Springer, pp. 283–296.

Bouffard, G., Thampi, B. N. & Lanet, J.-L. (2013), Detecting Laser Fault Injection for Smart Cards Using Security Automata, *in* S. M. Thampi, P. K. Atrey, C.-I. Fan & G. M. Pérez, eds, 'SSCC', Vol. 377 of *Communications in Computer and Information Science*, Springer, pp. 18–29.

Dubreuil, J., Bouffard, G., Thampi, B. N. & Lanet, J.-L. (2013), 'Mitigating Type Confusion on Java Card', *IJSSE* **4**(2), 19–39.

Farissi, I. E., Azizi, M., Moussaoui, M. & Lanet, J.-L. (2013), Neural network Vs Bayesian network to detect javacard mutants, *in* 'Colloque International sur la Sécurité des Systèmes d'Information (CISSE)', Kenitra, Marocco.

Girard, P., Villegas, K., Lanet, J.-L. & Plateaux, A. (2010), A new payment protocol over the Internet, *in* 'CRiSIS', IEEE, pp. 1–6.

GlobalPlatform (2011), *Card Specification*, 2.2.1 edn, GlobalPlatform Inc.

Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A. & Reygnaud, A. (2012), Subverting Byte Code Linker service to characterize Java Card API, *in* 'Seventh Conference on Network and Information Systems Security 2012', pp. 75–81.

Hemme, L. (2004), A Differential Fault Attack Against Early Rounds of (Triple-)DES, *in* M. Joye & J.-J. Quisquater, eds, 'CHES 2004', Vol. 3156 of *Lecture Notes in Computer Science*, Springer, pp. 254–267.

Joye, M., Quisquater, J.-J., Bao, F. & Deng, R. H. (1997), RSA-type Signatures in the Presence of Transient Faults, *in* M. Darnell, ed., 'IMA Int. Conf.', Vol. 1355 of *Lecture Notes in Computer Science*, Springer, pp. 155–160.

Jr., B. S. K., Çetin Kaya Koç & Paar, C., eds (2003), *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, Vol. 2523 of *Lecture Notes in Computer Science*, Springer.

Lancia, J. (2012), Java Card Combined Attacks with Localization-Agnostic Fault Injection, *in* Mangard (2013), pp. 31–45.

Mangard, S., ed. (2013), *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, Vol. 7771 of *Lecture Notes in Computer Science*, Springer.

McDougall, M., Alur, R. & Gunter, C. A. (2004), A model-based approach to integrating security policies for embedded devices, *in* G. C. Buttazzo, ed., 'EMSOFT', ACM, pp. 211–219.

Oracle (2011), *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*, Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Pavlova, M., Barthe, G., Burdy, L., Huisman, M. & Lanet, J.-L. (2004), Enforcing High-Level Security Properties for Applets, *in* J.-J. Quisquater, P. Paradinas, Y. Deswarte & A. A. E. Kalam, eds, 'CARDIS', Kluwer, pp. 1–16.

Piret, G. & Quisquater, J.-J. (2003), A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD, *in* C. D. Walter, c. K. Koçn & C. Paar, eds, 'CHES 2003', Vol. 2779 of *Lecture Notes in Computer Science*, Springer, pp. 77–88.

Prevost, S. & Sachdeva, K. (2006), 'Application code integrity check during virtual machine runtime'. US Patent App. 10/929,221.

Razafindralambo, T., Bouffard, G., Thampi, B. N. & Lanet, J.-L. (2012), A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks, *in* S. M. Thampi, A. Y. Zomaya, T. Strufe, J. M. A. Calero & T. Thomas, eds, 'SNDS', Vol. 335 of *Communications in Computer and Information Science*, Springer, pp. 185–194.

Schneider, F. B. (2000), 'Enforceable security policies', *ACM Trans. Inf. Syst. Secur.* **3**(1), 30–50.

Skorobogatov, S. P. & Anderson, R. J. (2002), Optical Fault Induction Attacks, *in* Jr. et al. (2003), pp. 2–12.

Wagner, D. (2004), Cryptanalysis of a provably secure CRT-RSA algorithm, *in* V. Atluri, B. Pfitzmann & P. D. McDaniel, eds, 'ACM Conference on Computer and Communications Security', ACM, pp. 92–97.