Fuzzing and Overflows in Java Card Smart Cards

Julien Lancia and Guillaume Bouffard

[1] THALES Communications and Security S.A.S
Parc technologique du canal, Campus 2 – Bat.A
3 avenue de l'Europe, 31400 Toulouse, France
julien.lancia@thalesgroup.com
[2] Agence Nationale de la Sécurité des Systèmes d'Informations (ANSSI),
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.
guillaume.bouffard@ssi.gouv.fr

Abstract. The Byte Code Verifier (BCV) is one of the most important security element in the Java Card environment. Indeed, embedded applets must be verified prior installation to prevent ill-formed applet loading. At the CARDIS 2015 conference, we disclosed a flaw in the Oracle BCV which affects the applet linking process and can be exploited on real world Java Card smart cards. In this article, we present how this vulnerability had been found and our exploitation of this flaw on a Java Card implementation that enables injecting and executing arbitrary native malicious code in the communication buffer from a verified applet. This attack was evaluated on several Java Card implementations with black box approach. In this case, as we cannot evaluate the effect of the control flow redirection caused by the attack, we develop a generic function which can be executed from any point.

Key words: Java Card, Software Attack, BCV vulnerabilities

1 Introduction

Developing smart card applications is a long and complex process. Despite existing standardization efforts, *e.g.*, concerning power supply, input and output signals, smart card development used to rely on proprietary Application Programming Interfaces (APIs) provided by each manufacturer. The main drawback of this development approach is that the code of the application can only be executed on a specific platform, thus lowering interoperability.

To improve the interoperability and the security of embedded softwares, the Java Card technology was designed in 1997 to allow Java-based applications for securely running on smart cards and similar footprint devices. Due to the resources constraints of this device, only a subset of the Java technology was retained in the Java Card technology. The

trade-offs made on the Java architecture to permit embedding the Java Card Virtual Machine (JCVM) on low resource devices concern both functional and security aspects.

1.1 The Java Card Security Model

In the Java realm, some aspects of the software security relie on the Bytecode Verifier (BCV). The BCV guarantees type correctness of the code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in Java to perform arithmetic operations on references. Thus, it must be proved that the two elements on top of the stack are bytes, shorts or integers before performing any arithmetic operations. Because Java Card does not support dynamic class loading, bytecode verification is performed at loading time, *i.e.* before installing the Converted APplet (CAP) file onto the card. Moreover, most of Java Card platforms do not embed an on-card BCV as it is expensive in terms of memory consumption. Thus, bytecode verification is performed off-card, either directly by the card issuer if he masters the loading chain, or by a trusted third party that signs the application as a verification proof.

In addition to static off-card verification enforced by the BCV, the Java Card Firewall performs runtime checks to guarantee applets isolation. The Firewall partitions Java Card's platform into separated protected object spaces called contexts. Each package is associated to a context, thus preventing instances of a package from accessing (reading or writing) data of other packages, unless it explicitly exposes functionality through a Shareable Interface Object.

Despite all the security features enforced by the Java Card environment, several attack paths [3, 4, 6–8, 16, 17, 20, 21, 23, 28, 32] have been found exploitable by the Java Card security community.

1.2 State-of-the-art on Java Card Bytecode Verifier flaws

The BCV is a key component of the Java Card platform's security. A single unchecked element in the CAP file, while apparently insignificant, can introduce critical security flaws in smart cards as shown in [17].

Although exhaustively testing a piece of software is a complex problem, several attempts have been made to characterize the BCV of the Java Standard Edition from a functional and security point of view. In [34], the authors rely on automatic test cases generation through code mutation and use a reference Virtual Machine (VM) implementation including a

BCV as oracle. In [11], a formal model of the VM including the BCV is designed, then model-based testing is used to generate test cases and to assess their conformance to the model.

In the Java Card community, several works aim at providing a reference implementation of an off-card [25] or an on-card [5, 14] Java Card BCV. These implementations are mainly designed from a formal model and can be used to test the BCV implementation provided by Oracle. As for the VM, model-based testing approaches [10, 33] were used to assess on Java Card BCV implementations. As of today, no full reference implementation or model of the Java Card BCV has been proposed.

The Oracle's BCV implementation in version 2.2.2 was analyzed by Faugeron et al. [17]. In this implementation, the authors identified an issue in the branching instructions interpretation during the type-level abstract interpretation performed by the BCV. The authors exploited this issue to perform a type confusion in a local variable, undetected by Oracle's BCV. This issue in the BCV was patched by Oracle from version 3.0.3.

Since the version 3.0.3, no security flaw identification or exploitation in the Java Card BCV has been publicly signaled. In this paper, we come back to a flaw discovered in the Java Card BCV from version 2.2.2 to 3.0.5 and we describe an exploitation of this flaw. This vulnerability was first disclosed at the CARDIS 2015 [24] conference. This article will introduce how this vulnerability had be found, based on a fuzzing approach. After evaluating this attack on several Java Card smart cards, a method to characterize the control flow transfer was developed.

Section 2 introduces our fuzzer and how a missing check in the Oracle's BCV implementation may allow an adversary to control a method offset and thus to trigger unverified bytecode execution. Section 3 shows how to succeed in exploiting this mechanism on a real Java Card product to trigger the execution of native code injected in a communication buffer. Finally, we evaluate our results on other Java Card products, define a generic method to characterize the control flow transfer with the black box approach and propose a countermeasure to prevent the attack.

2 A Flaw in the BCV

2.1 The BCV Duty

The BCV enforces various security and consistency checks that guarantee each embedded application remains confined in its own sandbox.

These verifications are performed on the CAP file, which is the binary representation of the classes that are loaded on the card.

The CAP file verification is performed in several passes. Passes 1 and 2 check that the format of the CAP file is consistent with the JCVM specification [30], excluding the portions of the archive that contain the methods bytecodes. Pass 3 performs a symbolic execution of the methods bytecodes to ensure type correctness. Eventually, pass 4 checks that symbolic references from instructions to classes, interfaces, fields, and methods are correct.

2.2 Verification of the CAP File Structure

The CAP file is composed of twelve different components, with internal and external dependencies, that are checked during the CAP file verification. Internal dependencies verification aims at validating the component properties as defined by the Java Card specification. External dependencies checks validate that redundant information specified in different components are compliant with each other. For example, each component has a `size` field that must be compliant with the `component-sizes` array contained in the Directory component where the sizes of every components are specified. An overview of all external dependencies between components in a CAP file are summarized in figure 1 borrowed from [19].

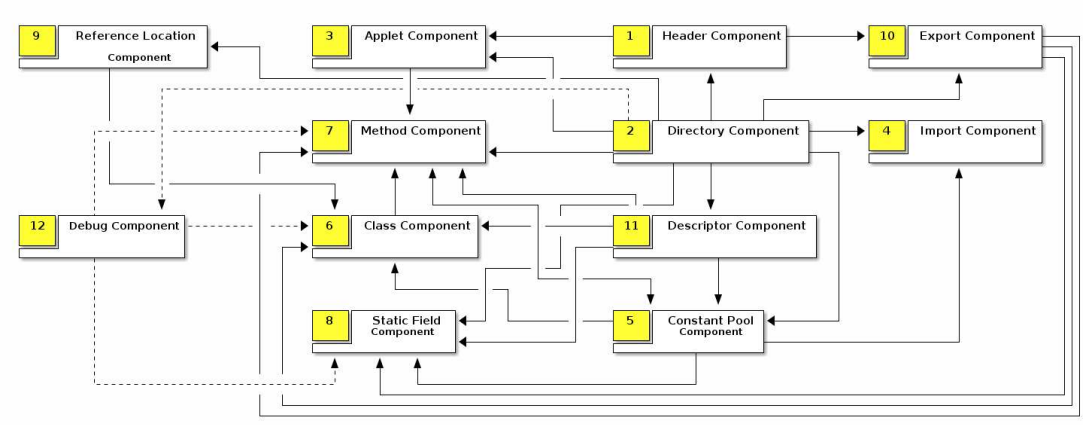

Fig. 1. External dependencies between components in a CAP file [19].

Among the twelve components stored in the CAP file, we will focus on the following components:

- the Method component stores the code of all methods in the package, concatenated as a set of bytes;
- the Constant Pool component contains an entry for each of classes, methods and fields referenced in the Method component;
- the Class component describes each classes and interfaces defined in the package, in a way that allows executing operations on that class or interface;
- the Descriptor component provides sufficient information to parse and verify all elements of the CAP file. This component is the main entry point for a bytecode verification.

The Descriptor component is the keystone of the BCV operations, but it has little or no importance for the card's processing and is therefore optionally provided during the loading of the applet.

Because of its purpose, the Descriptor component references several elements in the other components, and even provides redundant information with regards to these components. On the opposite, no component references the Descriptor component.

Considering the complex structure of the CAP file, parsing its structure for verification purpose is error prone. In order to identify flaws in the CAP file verifier, we applied a testing technique known for its good results on complex files and protocols structures, the fuzzing technique.

2.3 Fuzzing the bytecode verifier

Introduction to fuzzing Fuzzing is a simple and efficient technique to identify defects in software implementations as for example in [1, 2, 13, 18, 22, 36, 37]. It is generally used as a black box testing approach in which test cases are automatically generated and submitted to the system under testing in order to stress its robustness. This testing technique allows exploring a large amount of tests in an automatic way, what would be fastidious to do manually. Generally speaking, a fuzzer is composed of three main functions:

- data generation: create the data that will be sent to the tested target,
- data transmission: send the data to the tested target,
- target monitoring and logging: detect and record target anomalies.

We use the fuzzing testing methodology to seek vulnerabilities in the implementation of the bytecode verifier. As the BCV enforces all the requirements imposed by the JCVM specification, our work aims at discovering specification violations undetected by the BCV. Although other

works have already explored the automated test generation approach to discover flaws in the bytecode verifier through fuzzing and grammar based generation [12, 35], these works are mainly focused on the passes 3 (symbolic execution) and 4 (symbolic references checks) of the bytecode verification that guarantee validity of instruction type and instructions references. They are basically confined to the method's bytecode components of the CAP file. However, the structure of a CAP file is complex, and its structural correctness is fundamental for a correct execution of the code. In order to highlight the complexity of the CAP file structure, and to motivate the need of fuzzing testing on all elements of the archive beyond the bytecodes, we present a short example of dependencies that exist between the different components of a CAP file.

Applying fuzzing technique to the Bytecode Verifier We designed a CAP file fuzzer whose fuzzing method is inspired by genetic algorithms. It mimics genetic mutations and natural selection to find relevant test cases, a technique also known as evolutionary fuzzing [9, 15, 38]. In our BCV fuzzer, each test case is metaphorically equivalent to an individual in a population's generation. The JCVM behaviors caused by a test case are fully defined by the CAP file used for this test case, the same as the phenotype of an individual is fully defined by its DNA. Continuing along the genetic metaphor, we consider the bytes composing the CAP file as the nucleotides of the DNA of an individual. Our fuzzing approach is the same as the evolution of a race along the generations. Evolution relies on mutations that occur on the DNA of the individuals of a generation, creating new combinations of nucleotides (*i.e.* genes). Along a process called natural selection, mutations that fit the most the hostile environment (*e.g.*, predators) are conserved in the population and can be transmitted to the next generation, thus improving adaptation of the race to its environment. Similarly, in our BCV fuzzer, fuzzing is performed through mutation of the CAP file sequence. We implement the three main point mutations that occur in DNA mutation: insertion, deletion and transversion. The mutations perform the following modifications on the CAP file:

- insertion: insert a byte in the CAP file,
- deletion: delete a byte in the CAP file,
- transversion: modify the value of a byte in the CAP file.

During insertion and deletion mutations, only a subset of all mutations (composed of some remarkable and random values) is used to prevent combinatorial explosion. The fuzzer also preserves high level grammatical

constraints on the CAP file (*e.g.*, update of array size when performing insertion mutation on an array). Consequently, mutations affect only a targeted portion of the CAP while the overall structure remains correct. The natural selection is performed by a single predator, the BCV. Only mutations that create a valid CAP file according to the Java Card BCV are allowed to carry onto the next generation, and to pursue the fuzzing process by mutation. All the valid mutated CAP files are then executed in a simulator, looking for mutated CAP files that triggers an unexpected behavior of the JCVM (crash, unexpected data output). These CAP files are hand-analyzed, along with the bug reports, to determine the origin of the misbehavior.

An analysis of the bug reports generated by the BCV fuzzer brought us to identify a missing external dependency check between the Class component and the Descriptor component. We present the details of this BCV flaw and the resulting exploitation in the next sections.

2.4 Missing Check in the BCV

The missing check we have identified in the BCV involves the token-based linking scheme. This scheme allows downloaded software to be linked with API already embedded on the card. Accordingly, each externally visible item in a package is assigned a public token that can be referenced from another package. There are three kinds of items that can be assigned public tokens: classes, fields and methods. The bytecodes in the Method component refer to the items in the Constant Pool component, where the tokens required to perform the bytecode operation (*e.g.*, class and method token for a method invoke) are specified.

When the CAP file is loaded on the card, the tokens are linked with the API and resolved to the internal representation used by the VM. The linking process operates on the bytecode and is performed in several steps:

1. each token is an index in the Constant Pool component. The item stored at the provided index specifies the public tokens of the required items (*e.g.*, class and method token for a method invoke);
2. the tokens are resolved into the JCVM internal representation. For a method invoke, the class token identifies a `class_info` element in the Class component;
3. in the `class_info` element, the `public_virtual_method_table` array stores the methods internal representation. The method token is an index into the `public_virtual_method_table` array;

4. the element in the `public_virtual_method_table` at the method token index is an absolute offset in the Method component to the header and the bytecode of the method to execute.

   The figure 2 summarizes the linking process for a method call.

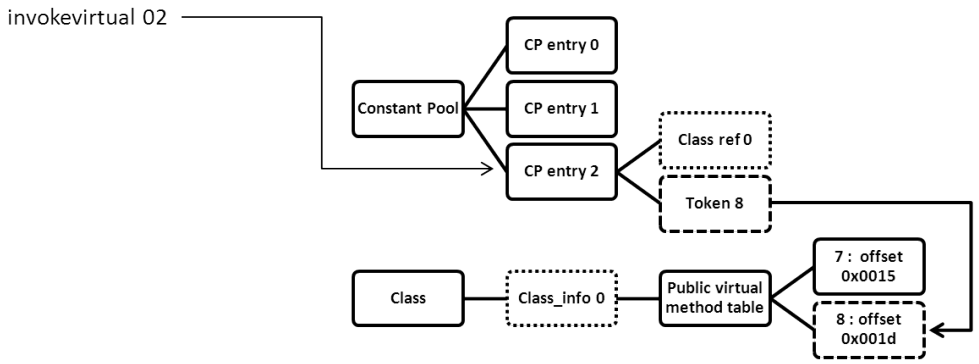

Fig. 2. Overview of the linking process for a method call.

The absolute offset in the Method component to the header and the bytecode of the method to execute is a redundant information in the CAP file as it is stored both in the `public_virtual_method_table` elements in the Class component and in the `method_descriptor_info` elements in the Descriptor component. The offset information in the Descriptor component is used exclusively by the BCV before loading, while the offset information in the Class component is used exclusively by the JCVM linker on card. Thus, any ill-formed offset information in the Class component remains undetected by the BCV checks, but is still used by the JCVM linker on card.

2.5 Exploiting the BCV flaw

As presented so far, the BCV flaw we expose allows manipulating the method offset information in the Class component while remaining consistent with the BCV checks. The exploitation of this flaw consists in deleting an entry in the `public_virtual_method_table` of a `class_info` element in the CAP file. The resolution of the corresponding method offset during the JCVM linking leads to an overflow in the Class component, as presented in figure 3. This overflow brings the JCVM to interpret the content of the memory area following the Class component on card as a method index.

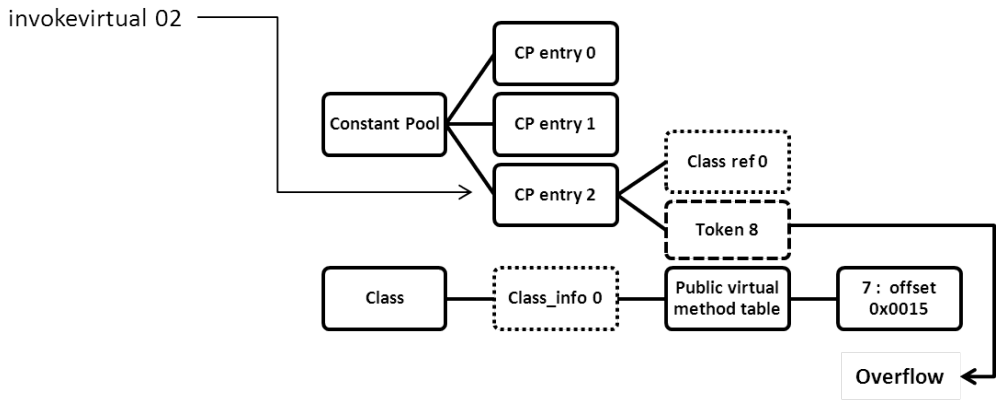

Fig. 3. Overflow in the linking process with for a method call.

The loading order of the CAP components is defined by the Java Card specification. This order specifies that the Method component is loaded right after the Class component. It is thus very likely that the Method component is stored next to the Class component in the card's memory. As a result, the Class component overflow is likely to fall into the Method component. In this eventuality, the offset of the method resolved in overflow is the numerical value of a bytecode in the Method component, that can be controlled by the applet developer. The figure 4 presents an exploitation of the Class component overflow through the Method component.

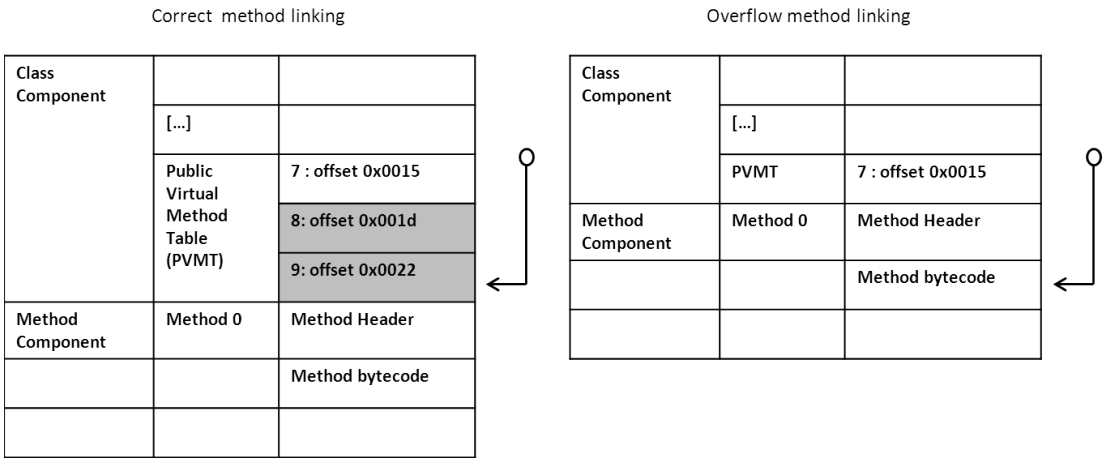

Fig. 4. Figure on left shows a successful linking in the Class component. Figure on the right shows the Class component overflow during linking when grayed out elements are deleted. Class component overflow falls into the Method component.

3 Code Injection from a Bytecode Verified Applet

In the previous section, we have presented, in the eventuality of a favorable memory mapping, how an attacker can exploit a BCV flaw to specify an arbitrary method offset in a BCV validated applet. In this section, we present the exploitation of this flaw on a real product that allows us to inject and execute native code in a communication buffer from a BCV validated applet.

The attack steps necessary to reach arbitrary native code on the Java platform are summed up hereafter and detailed in the next sections. First, we exploit the BCV flaw presented in section 2 to forge an arbitrary method header in the Method component. This arbitrary method header is then used to abuse the native method execution mechanism of the platform and thus create a buffer overflow in the native method table. Finally, this buffer overflow allows dereferencing the communication buffer address as a native function. As a consequence, the data sent to our verified applet through the communication channel are executed as native code on the JCVM.

This full attack is a proof of concept to demonstrate that the flaw discovered in the Oracle BCV may jeopardize the security of Java Card smart cards.

3.1 Native Execution in the Virtual Machine

We validate the exploitation of the BCV flaw on an open Java Card platform embedded on an ARM micro-controller. This Java Card platform was provided in the context of a security expertise, thus both the code and the memory mapping of the VM were made available.

The runtime environment of this platform provides a mechanism that allows switching execution to native implementations of Java Card API methods for performance reasons. The implementation of this mechanism is similar to the Java Native Interface (JNI) mechanism provided in classical Java VMs [27].

In the JNI approach, the native methods are identified through a dedicated flag (`ACC_NATIVE`) in the method header. According to the JCVM specification, the native header flag is only valid for methods located in the card mask. Therefore a native method loaded in a CAP file is not compliant with the Java Card specification, and is thus rejected by the off-card verifier.

The native method resolution in JNI relies on *interface pointers*. An interface pointer is a pointer to a pointer. This pointer refers to an array

of pointers, each one itself pointing on to an interface function. Each interface function is stored at a predefined offset inside the array. Figure 5 illustrates the organization of an interface pointer. The offset inside the array where the native function pointer is to be found is provided in the body of the native method.

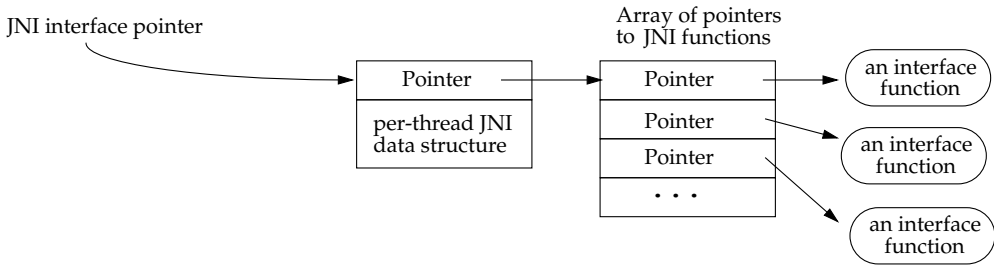

Fig. 5. JNI functions and pointers [26].

3.2 Native execution from a validated applet

As presented in section 2, a missing check in the BCV can cause an overflow that brings the VM to resolve the method offsets outside the Class component. In the VM implementation, we use to exploit our attack, the Class component overflow falls into the Method component so the value of the method offset can be specified as the numerical value of a bytecode in the Method component.

According to the JCVM specification [30], the offset of a method must point to a method header structure in the Method component, followed by the bytecode of the method. When exploiting the BCV flaw, the offset is controlled by the developer so it can point to any portion of the Method component. This can be used to make the method offset pointing on a portion of the bytecode that can be interpreted as a method header. The `iipush` bytecode can be used for this purpose, as its operand is a 4-bytes constant that is not interpreted by the BCV. This 4-bytes constant is thus used to code a method header containing the `ACC_NATIVE` flag and the native method index. This `iipush` bytecode is accepted by the BCV because it forms a valid bytecode sequence, but when the operand is interpreted as a native method header (through an overflow on the Class component), the control flow switches to native execution. Figure 6 shows the attack path from the Class component overflow to the native execution of a JNI method.

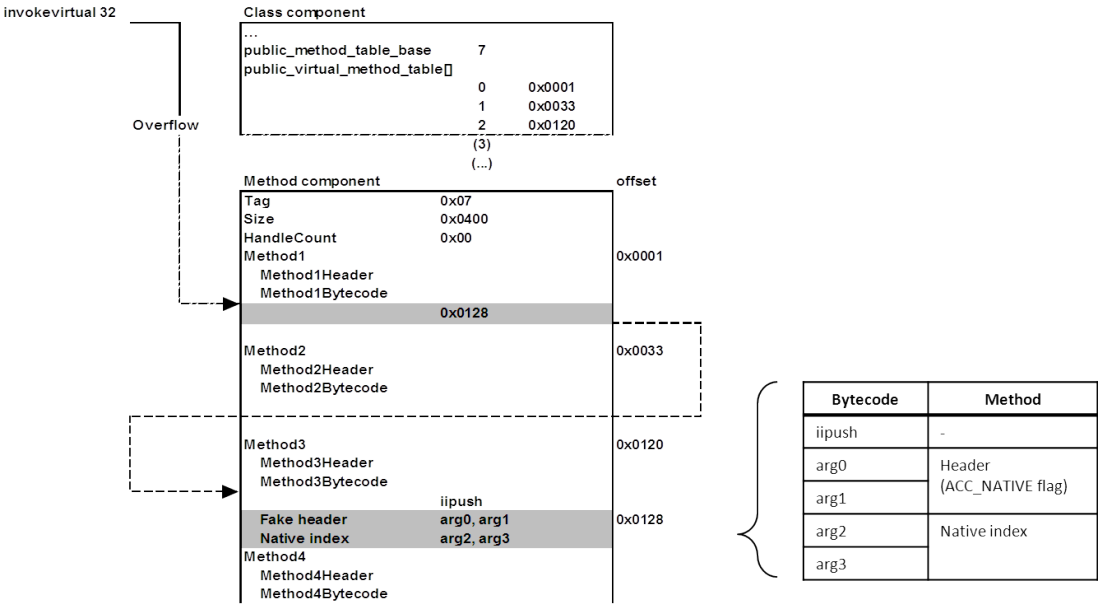

Fig. 6. Exploitation of the Class component overflow to execute a native method.

We were thus able, by specifying the adequate value for the method offset, to execute any of the native methods provided by the VM in the array of JNI native function pointers.

3.3 Abusing the Native Execution Mechanism for Code Injection

The attack so far allows calling JNI native methods provided by the platform, that are stored in an array of JNI native function pointers (or *native array*). When a native method call occurs, the switch from the Java runtime environment to the native execution environment requires an index in the *native array* to determine the native function pointer. Experimentation on the target JCVM allowed us determining that an overflow on the *native array* can be achieved by specifying the relevant index in the native method body. Thus, any memory content stored next to the *native array* can be exploited as a native function pointer.

An analysis of the memory mapping of the product shows that a memory zone next to the *native array* contains a pointer to the communication buffer used for Host Controller Protocol (HCP) communications. The HCP protocol handles the transport layer of the Single Wire Protocol (SWP) protocol, involved in Near Field Communication (NFC) communications with smart cards. HCP messages encapsulates ISO7816

Application Protocol Data Unit (APDU) that are conveyed to the smart card over SWP.

Using the overflow on the *native array*, we are able to use the HCP communication buffer pointer as native function pointer. The execution of this native function pointer leads to executing the content of the HCP communication buffer as a native assembly function.

The HCP protocol has several properties that limit the use of the HCP communication buffer as a native payload injection placeholder:

1. HCP packets are prefixed with a HCP message header and an HCP packet header. These headers are interpreted as native assembly opcodes.
2. HCP enforces fragmentation of messages, which limits packets size to 27 bytes. The entire native payload must thus be contained in 27 bytes.

In order to gain more space to inject our attack payload, we inject a minimal payload in the HCP communication buffer whose only purpose is to redirect the execution flow to the ISO7816 APDU buffer. This minimal redirection payload is presented in Table 1. Because the HCP communication buffer pointer is used as a function pointer, all the HCP buffer is interpreted as native code, including packet header, message header and encapsulated APDU header. These header bytes produce no side effect as shown in Table 1, which lets the redirection payload execute properly.

HCP message	Interpretation	Native code	Comment
82 50	Packet header Message header	`STR r2, [r0, r2]`	No side effect
00 10	CLA/INS	`ASRS r0, r0, #0`	No side effect
00 00	P1/P2	`MOVS r0, r0`	No side effect
14 00	Lc/padding	`MOVS r4, r2`	No side effect
E9 2D 5F FC	Data	`PUSH {r2-r12, lr}`	
F6 4A 54 D0		`MOVW r4, #0xADD0`	
F6 CA 54 D1		`MOVT r4, #0xADD1`	`r4 = &apduBuffer`
47 A0		`BLX r4`	`branch to apduBuffer`
E8 BD 9F FC		`POP {r2-r12, pc}`	

Table 1. Native payload in the HCP buffer that redirects the execution flow to the APDU buffer. Relevant payload data is grayed out.

The ISO7816 protocol has broaden fragmentation constraints, which offers sufficient space for a full native payload injection. We present in

Table 2 a full payload injected in the APDU buffer that branches to a low level read/write OS function. Because the start address execution is chosen from the HCP message buffer payload, the header bytes are skipped and the native execution starts at the `push` instruction (Table 2, 3rd row).

	APDU		Native code	Comment
1	00 12 00 00 31	Header		CLA/INS/P1/P2/Lc
2	B1 FA 15 00	Data		source reading address
3	2D E9 FF 5F		PUSH {r0-r12, lr}	
4	F6 4A 56 D0		MOVW r6, #0xADD0	
5	F6 CA 56 D1		MOVT r6, #0xADD1	r6 = apduBuffer
6	35 68		LDR r5, [r6,#0x00]	r5 = *apduBuffer
7	28 46		MOV r0, r5	
8	00 F1 09 00		ADD r0, r0, #0x6A	*dest: apduBuffer + 0x6A
9	D5 F8 05 10		LDR r1, [r5, #0x08]	*src: *(apduBuffer + 8)
10	4F F0 40 02		MOV r2, #0x40	length: 0x40
11	F6 4A 54 D2		MOVW r4, #0xADD2	
12	F6 CA 54 D3		MOVT r4, #0xADD3	r4 = *read_function_ptr()
13	A0 47		BLX r4	call method
14	BD E8 FF 9F		POP {r0-r12, pc}	

Table 2. Native payload in the ISO7816 APDU buffer that calls an OS function to read an arbitrary memory zone and copy the result to the APDU buffer. Relevant payload data is grayed out.

The payload initializes the source parameter to the first 4 bytes of the payload (Table 2, 2nd row), such that the reading address can be selected directly in the APDU. Then, it initializes the destination address (where the read bytes are copied) to the address of the APDU buffer following the payload, such that the read bytes are immediately available for sending back through the APDU buffer. Finally, it branches to the low level OS function that performs the reading operation. As a result, any physical address of the card can be accessed through this native payload.

Figure 7 shows the execution flow from the *native array* overflow to the redirection payload in the HCP message buffer to the final attack payload in the APDU buffer. We were able to integrally dump the card memory and to reverse it using commercial reversing tools. The reversed code was identified as the code of the embedded JCVM.

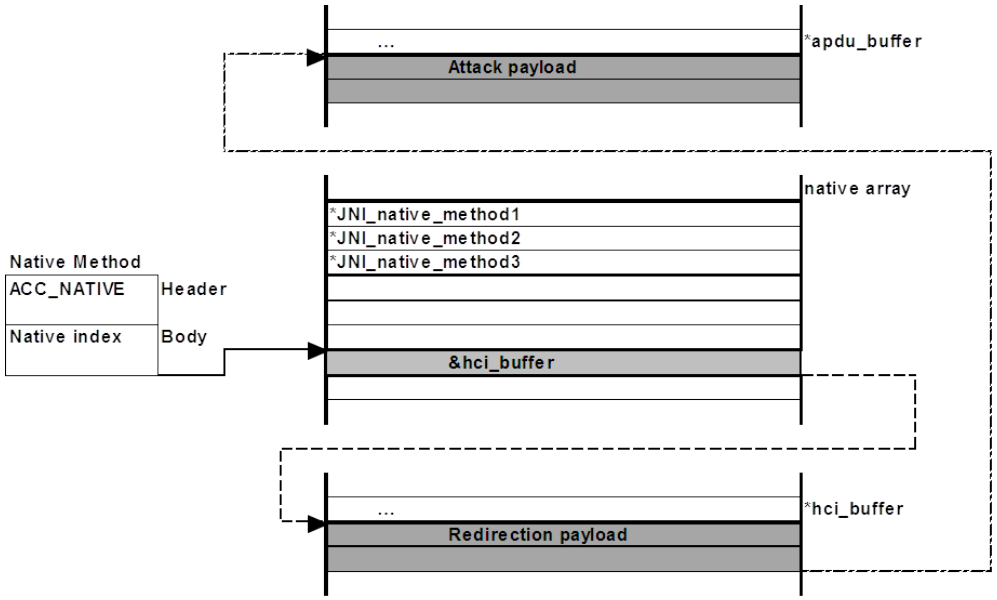

Fig. 7. Exploitation of the *native array* overflow to execute native code in the APDU buffer.

4 Other Experimental Results

To evaluate the consequence of the BCV flaw on a broader range of virtual machine implementations, we tested, on different smart cards from different manufacturers, how much each of them supports the installation of an ill-formed applet. We evaluated seven cards from three distinct manufacturers (a, b and c). Each card name is associated with the manufacturer reference and its Java Card specification [30]. The list of evaluated Java Card smart cards is presented in table 3.

Reference	Java Card Platform	GlobalPlatform Version	Details
a-22a	2.2.1	2.1.1	36 kB EEPROM, RSA
a-22b	2.2.2	2.1.2	80 kB EEPROM, RSA
a-30c	3.0.4	2.2.1	80 kB EEPROM, ePassport
b-30a	3.0.1	2.2.1	1 MB Flash memory, (U)SIM
c-21a	2.1.1	2.0.1	128 kB EEPROM, SIM
c-21b	2.1.1	2.0.1	64 kB EEPROM, RSA, AES
c-22c	2.2.2	2.2.2	256 kB Flash memory, (U)SIM

Table 3. Cards evaluated during this experimentation.

None of the evaluated card embeds an embedded BCV. On each card, an ill-formed applet can be installed and, if the installation succeeds, the

applet is executed. The ill-formed applet has a dereferenced method in the public virtual methods table. Table 4 sums up the cards reactions.

Ref.	Statut
a-22a	PCSC error: card mute. ✗
a-22b	PCSC error: card mute. ✗
a-30c	PCSC error: card mute. ✗
b-30a	No error: the card return the value 0x0701. ✗
c-21a	Global platform error: error during the loading process (*applet rejected*). ✓
c-21b	Global platform error: error during the loading process (*applet rejected*). ✓
c-22c	Global platform error: error during the loading process (*applet rejected*). ✓

Table 4. Statuts of each evaluated cards.

As shown in table 4, cards react differently to the ill-formed CAP file installation and execution. The cards with the symbol (✓) detect the ill-formed CAP file during the installation and reject it. On the other cards, marked with the symbol (✗), installation and execution succeed.

Successful executions cause either unexpected card response or card mute. Unexpected card response indicates that unexpected code execution occurred. Card mute may result from infinite loop or card's reaction to illegal code, which also indicates unexpected code execution.

These behaviors proof that the control flow of the JCVM is modified. We can thus conclude that the BCV flaw presented in this article can be exploited on a range of different Java Card smart cards.

The full attack path that results in arbitrary native code execution requires information about the memory mapping and the JCVM implementation that were not available for these tests. Therefore, we did not attempt to reproduce the full attack path.

4.1 Characterizing the Control Flow Transfer

The attack presented in section 2.5 exploits an overflow that occurs during the linking process and forces the program counter to an incorrect value during execution. As a result, the control flow is transferred to an incorrect memory zone. The exploitation of the full attack path requires knowning where the control flow is transferred thus we designed a characterization process that allows an attacker determining the jump target location resulting from the offset overflow.

During the linking process, the offset of the method to resolve is sought in the `public_virtual_method_table` of a `class_info` element in the

`Class` component of the CAP file. When an attacker deletes the required offset in the CAP file, the offset is taken in overflow in the next memory bytes. As the attacker can delete as many offsets as there are methods in the CAP file, several bytes of overflow memory can be used as method offset.

To characterize the control flow resulting from an unknown offset we design a method that can be executed from any point, regardless of where the erroneous control flow has jumped. To reach this goal, each element of the method should be interpretable both as a method header and a Java Card instruction. If the overflow offset does not jump in the characterization method, another offset is deleted in the CAP file and thus the next bytes in memory are used as offset. The process is repeated until the jump target place is located in the characterization method. Because offset are signed values, the `Method` component can be located in memory either before or after the `Class` component.

Constraints of a Java Card Method From the Java Card specification [30], we extracted the constraints which define a valid Java Card method.

A Java Card method is composed of two elements, a header and a set of instructions which form the method's bytecode. According to the Java Card specification, the header is a structure defined as a `method_header_info` or an `extended_method_header_info`. The Listing 1 shows those structures, where the type `u1` defines an unsigned byte (8-bit length) and the type `bit[4]`, a 4-bit item.

```
method_header_info {
  u1 bitfield {
    bit[4] flags
    bit[4] max_stack
  }
  u1 bitfield {
    bit[4] nargs
    bit[4] max_locals
  }
}
```

```
extended_method_header_info {
  u1 bitfield {
    bit[4] flags
    bit[4] padding
  }
  u1 max_stack
  u1 nargs
  u1 max_locals

}
```

Listing 1. Java Card method defines from the JCVM specification.

As presented in the Listing 1, a method header is defined by the following elements:

– The `flags` item is a mask of modifiers which define this method – standard, extended or abstract method. The native method is not supported by the JCVM specification [30].

- The `max_stack` item indicates the maximum number of words required on the operand stack during execution of this method.
- The `nargs` item represents the number of words required to represent the parameters passed to the method.
- The `max_local` item indicates the number of words required to represent the local variables declared by this method without the method parameters.

The method instructions are composed of opcodes, encoded on 1 byte, and a set of 1-byte arguments. The valid opcode values are comprised between `0x00` (`NOP`) and `0xB8` (`putfield_i_this`). Values between `0xB9` and `0xFD` are reserved for future use and cannot be used in compliant JCVM implementations. The opcode values `0xFE` and `0xFF` are intended to provide *back doors* or traps to implementation-specific functionality implemented in software and hardware, respectively [30].

We extracted and factorized all method constraints defined by the JCVM specification, and we obtained the model shown in the Listing 2.

```
method = header & bytecodes

header = method_header_info | extended_method_header_info
method_header_info = {
  // method_header_info has a 4-bit element.
  flags      = {0x0, 0x4}, // 0x0: a  standard method
                           // 0x4: an abstract method
  max_stack  = [0x0, 0xF],
  narg       = [0x1, 0xF] // [0x0, 0xF] for static method
  max_local  = [0x0, 0xF]
}
extended_method_header_info = {
  // extended_method_header_info has a 1-byte element
  flags      = {0x8, 0xC},// 0x8: an extended method
                          // 0xC: an extended abstract method
  padding    = 0x0,
  max_stack  = [0x00, 0xFF],
  nags       = [0x01, 0xFF], // [0x00, 0xFF] for static method
  max_local  = [0x00, 0xFF],
}

if (flags & EXTENDED_METHOD) // Is it an extended method?
  bytecodes = {} // empty set
else
  bytecodes = instruction+ // + = one or more

instruction = opcode & (argument)* // * = zero or more
opcode   = [0x00, 0xB8] // Reserved values from 0xB9 to 0xFF
argument = [0x00, 0xFF]
```

Listing 2. Constraints which define a Java Card method.

Solving the Constraints As the attacker cannot control the resolution of the `invokevirtual` instruction offset resulting from the overflow in the `Class` component, our characterization method aims at implementing a method which can be executed from any point. In this method, each byte can be interpreted as method header, an opcode or an argument. Indeed, as the `Class` component is generally located before the `Method` component in the card, an overflow from the `Class` component may reach the first method in the `Method` component.

Since the method constraints have been extracted from the Java Card specification, we look for a set of bytes which is a solution to these constraints.

Design of a polymorphic method From the method constraints defined in the Listing 2, we designed a method which can be executed from any point. To make each byte of the method interpretable as a valid execution entry point, they should be compliant with the following rules:

- Constraints on `method_header_info` ∪ constraints on `extended_method_header_info`
- with:
  - Constraints on `method_header_info` = [`0x01`, `0x0F`] ∪ [`0x11`, `0xFF`];
  - Constraints on `extended_method_header_info` = [`0x80`] ∪ [`0x01`, `0xFF`].

After minimizing the constraints, the following rule can we obtained:

$$\forall \text{ bytecode} \in \{[\texttt{0x01}, \texttt{0x0F}] \cup [\texttt{0x11}, \texttt{0xFF}]\}$$

To optimize the byte range value, we decided to exploit the exception mechanism. Indeed, the exception mechanism allows redirecting the execution flow to a finite set of handlers (exception handlers) from any point in the execution flow, which is the desirable behaviour for our polymorphic method.

The Listing 3 shows a java program implementing a set of exceptions thrown in the `try`-statement and caught in the appropriate `catch`-statement.

```
public void characterizedMethod(void) {
  try {
    // throw an exception();
    // throw an exception();
```

```
    // etc., several times
  } catch (NullPointerException npe)  {
    // Payload 1
  } catch (SecurityException se)      {
    // Payload 2
  } catch (Exception e)               {
    // Payload 3
  }
}
```

Listing 3. A Java Card method which throws and catches exceptions.

The function listed in the Listing 3 is compiled to the bytecode sequence shown in the Listing 4.

```
public void characterizedMethod(void) {
  01 // flag: 0 max_stack: 1
  01 // narg: 0 max_local: 1
  01       sconst_null
  93       athrow
  60 01    ifeq 01
  01       sconst_null
  93       athrow  // throw the exception.
  ...
  // Catches area
  ...
  7A       return // This bytecode is never reached.
}
```

Listing 4. A valid method bytecode which can be executed from any point.

In this bytecode sequence, the pattern `01 01 93 60` is repeated. Depending on the starting point in this sequence, different bytecode patterns can be ran. The Table 5 lists each pattern possibilities and the resulting exception thrown.

Sequence	Remark	Exception
`01 01 93 60`		`NullPointerException`
`01 93 60`	Empty stack	`SecurityException`
`93 60 01 01 93 60`	Invalid header	`SecurityException`
`60 01 01 93 60`	Invalid header	`SecurityException`

Table 5. Executed sequences and exceptions thrown.

The two last sequences, `93 60 01 01 93 60` and `60 01 01 93 60`, have an invalid header. Leading `9` and `6` nibbles cannot be valid header flags value according the Java Card specifications. However, several JCVM

implementations use a binary masks on the method headers to retrieve the relevant method's flags. Therefore, in this case the flag `9` may be interpreted as an extended method, and the flag `6` as an extended abstract method. According to the Java Card specification, for security reasons, the Java Card runtime environment implementation may also mute the card instead of throwing a `SecurityException` [29].

Finally, the attack payload is placed in the exception handlers. Thus, when an exception is thrown the execution flow is directed to the attack payload regardless of the execution starting point.

The approach introduced in this section aims at characterizing the effect of the overflow on the `public_virtual_method_table` field. For that purpose, we developed a polymorphic method which can be executed from any point. This polymorphic method, based on the java exception mechanism, transfers the execution flow to the exception handlers where the attack payload is located.

5 Conclusion, Countermeasure and Future Works

We show in this article how a missing check in the Oracle's BCV implementation can be exploited on a Java Card. This flaw was disclosed by an evolutionary fuzzer. We demonstrated that this BCV issue has a critical impact on smart cards security through a proof of concept exploitation on a JCVM implementation. We have successfully managed to inject and execute native code in a communication buffer, and finally gain full read/write OS privileges on the whole card memory. Finally, we evaluated on a range of different cards from different manufacturers that most of the JCVM implementations do not protect themselves against the BCV issue exploitation. As we evaluated cards in black box model, we faced the problem of characterizing the control flow transfer. To resolve this issue, we designed a polymorphic java method using method constraints. This method is semantically correct, regardless of the execution starting point, and redirects the execution flow to a single point where the attack payload is located.

Following our responsible disclosure of the BCV issue to Oracle, we were allowed to publish this article and a new version of the BCV was released[3]. This new BCV version detects the Class component inconsistency and thus mitigate our attack. A loading process including mandatory

[3] The BCV included in the Java Card SDK 3.0.5u1 prevents the introduced attack. This version was released on 19 August 2015.

bytecode verification step with the latest Oracle's BCV provides a valid countermeasure against the attack presented in this paper.

With the identification of a new flaw in the Oracle's BCV implementation, one sees that the BCV must be entirely verified to lower the risks of new vulnerabilities disclosure. To reach this objective, an effort should be done to specify the security and functional requirements a BCV must comply with in order to protect JCVM implementations against this software attack.

References

1. Pedram Amini. PaiMei-Reverse Engineering Framework. In *RECON06: Reverse Engineering Conference*, Montreal, Canada, 2006.
2. Pedram Amini and A Protnoy. Sulley fuzzing framework, 2010.
3. Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Prouff [31], pages 297–313.
4. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, pages 148–163, 2010.
5. Reinhard Berlach, Michael Lackner, Christian Steger, Johannes Loinig, and Ernst Haselsteiner. Memory-efficient On-card Byte Code Verification for Java Cards. In *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, CS2 '14, pages 37–40, New York, NY, USA, 2014. ACM.
6. Guillaume Bouffard. *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks*. PhD thesis, University of Limoges, Limoges, France, October 2014.
7. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In Prouff [31], pages 283–296.
8. Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a Java based smart card. *Computers & Security*, 50:33–46, 2015.
9. Sergey Bratus, Axel Hansen, and Anna Shubina. Lzfuzz: a fast compression-based fuzzer for poorly documented protocols. 2008.
10. Andrea Calvagna, Andrea Fornaia, and Emiliano Tramontana. Combinatorial Interaction Testing of a Java Card Static Verifier. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*, pages 84–87. IEEE Computer Society, 2014.
11. Andrea Calvagna and Emiliano Tramontana. Automated Conformance Testing of Java Virtual Machines. In Leonard Barolli, Fatos Xhafa, Hsing-Chung Chen, Antonio F. Gómez-Skarmeta, and Farooq Hussain, editors, *Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2013, Taichung, Taiwan, July 3-5, 2013*, pages 547–552. IEEE Computer Society, 2013.
12. Andrea Calvagna and Emiliano Tramontana. Combinatorial validation testing of Java card byte code verifiers. In *Enabling Technologies: Infrastructure for*

Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Workshop on, pages 347–352. IEEE, 2013.
13. Gabriel Campana. Fuzzgrind: an automatic fuzzing tool. *Hack.lu*, 2009.
14. Ludovic Casset. Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 2002.
15. Jared DeMott, Richard J. Enbody, and William F. Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. *BlackHat and Defcon*, 2007.
16. Emilie Faugeron. Manipulating the Frame Information with an Underflow Attack. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2013.
17. Emilie Faugeron and Sebastien Valette. How to hoax an off-card verifier. *e-smart*, 2010.
18. Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.
19. Samiya Hamadouche. Étude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité. Master's thesis, University M'Hamed Bougara of Boumerdes, Faculty of Sciences, LIMOSE Laboratory, 5 Avenue de l'indépendance, 35000 Boumerdes, Algeria, 2012.
20. Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire, and Arnaud Reygnaud. Subverting Byte Code Linker service to characterize Java Card API. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 2012.
21. Samiya Hamadouche and Jean-Louis Lanet. Virus in a smart card: Myth or reality? *Journal of Information Security and Applications*, 18(2-3):130–137, 2013.
22. Julien Lancia. Un framework de fuzzing pour cartes à puce: application aux protocoles emv. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, page 82, 2011.
23. Julien Lancia. Java Card Combined Attacks with Localization-Agnostic Fault Injection. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2012.
24. Julien Lancia and Guillaume Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany*, November 2015.
25. Xavier Leroy. Bytecode verification on Java smart cards. *Softw., Pract. Exper.*, 32(4):319–340, 2002.
26. Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, first edition edition, June 1999.
27. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, May 2014.
28. Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Gilles Grimaud and François-Xavier Standaert,

editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

29. Oracle. *Java Card 3 Platform, API Specification, Classic Edition*. Number Version 3.0.5. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, September 2011.
30. Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*. Number Version 3.0.5. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, 2015.
31. Emmanuel Prouff, editor. *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*. Springer, 2011.
32. Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard. In Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquín García-Alfaro, editors, *Data and Applications Security and Privacy XXVI - 26th Annual IFIP WG 11.3 Conference, DBSec 2012, Paris, France, July 11-13,2012. Proceedings*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer, 2012.
33. Aymerick Savary, Marc Frappier, and Jean-Louis Lanet. Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2013.
34. Emin Gün Sirer. Testing Java Virtual Machines. In *International Conference on Software Testing And Review*, San Jose, California, November 1999.
35. Emin Gün Sirer and Brian N Bershad. Testing Java virtual machines. In *Proc. Int. Conf. on Software Testing And Review*, 1999.
36. Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
37. Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.
38. Joachim Wegener. Evolutionary testing of embedded systems. In *Evolutionary Algorithms for Embedded System Design*, pages 1–33. Springer, 2003.