New Countermeasures against Fault and Software Type Confusion Attacks on Java Cards

Guillaume Barbu and Christophe Giraud

Oberthur Technologies
Cryptography & Security Group
4, allée du Doyen Georges Brus, 33 600 Pessac, France
{g.barbu,c.giraud}@oberthur.com

Abstract. Attacks based on type confusion against Java Card platforms have been widely studied in the literature over the past few years. Until now, no generic countermeasure has ever been proposed to cover simultaneously and efficiently direct and indirect type confusions. In this article we bridge this gap by introducing two different schemes which cover both type confusions. First, we show that an adequate random transformation of all the manipulated data on the platform according to their type can bring a very good resistance against type confusion exploits. Secondly, we describe how a so-called Java Card Virtual Machine *Abstract Companion* can allow one to detect all type confusions between integers and Objects all across the platform. While the second solution stands as a strong but resource-demanding mechanism, we show that the first one is a particularly efficient memory/security trade-off solution to secure the whole platform.

Keywords: Java Card, Countermeasures, Fault Injection, Combined Attacks.

1 Introduction

Putting Java into memory constrained embedded systems such as smartcards was not an easy bet. However, since its introduction in 1996, the Java Card technology has been successively adopted in all environments of the smartcard industry, from mobile telecommunication to banking and pay-tv, proving both the need for such a standard and the security level that can be reached on these platforms.

Actually, Java Cards can be considered as more secure than native cards due to the additional controls performed by the Java Card Runtime Environment. Among these, we can cite for instance the check of array boundaries or the application firewall ensuring the isolation between the different applications hosted by the card. In order to challenge the intrinsic security mechanisms of the platform, the notion of *ill-formed applications* has emerged. Ill-formed applications are obtained by modifying the binary representation of a Java Card

D. Naccache and D. Sauveron (Eds.): WISTP 2014, LNCS 8501, pp. 57–75, 2014.

application[1] in order to escape certain language rules or particular checks [1–5]. Hopefully, such logical attacks can be counteracted by the use of a *ByteCode Verifier* (BCV) [6, §4.9.2] which aims at ensuring that an application is conformed to the Java Card specifications [7–9]. Nevertheless, an attacker having hands on an open platform may not necessarily execute the BCV, unless it is embedded within the platform, as mandated for instance for the high-end Java Card *Connected* Edition [10–13] but rarely found on *Classic* platforms.

A few years after the publication of these logical attacks, the idea to combine malicious applications with fault injections emerged [14]. Firstly, *Fault Attacks* were known to affect the security of embedded cryptographic implementations, either asymmetric or symmetric [15–17]. But as stated in [18–20], these attacks can actually target any function implemented on embedded devices, which naturally includes Java Card internal routines. Nowadays, the main mean of disturbing an embedded chip is to use light beams [21] or electromagnetic pulses [22]. The so-called *Combined Attacks* aim at allowing a malicious but well-formed application (i.e. that passes the BCV) to bypass certain security features of Java Cards by using fault attacks. Since 2009, several Combined Attacks have been published to attack vital points of a Java Card such as the application firewall, the operand stack or the garbage collector [23–30].

In this paper, after recalling the main software and combined attacks, we analyze the countermeasures against type confusion presented so far in the literature. As a result of this analysis, we show not only that none of the proposed countermeasures can detect all kind of type confusions but also that some attack paths are not covered at all, even when combining these countermeasures. This statement is the motivation for our study towards a platform-wide approach to type confusion detection. We propose in the following two different solutions allowing one to cover the entire JCRE. The first one involves random transformations and comes with almost no memory overhead while offering *only* a probabilistic (although very close to 100 %) security level. On the other hand, the second proposal, entitled the Java Card Virtual Machine Abstract Companion, comes with a certain memory overhead while providing a very strong and deterministic security level. As we will see, these solutions allow to counteract all the identified attack paths and do not require any profound modification of existing Java Card implementations.

Section 2 introduces the different flavours of type confusions that can be encountered in the literature. Section 3 briefly describes the different countermeasures published so far and analyzes their benefits and flaws. Section 4 presents the two approaches we define and discuss their effectiveness and efficiency.

2 Type Confusion on Java Cards

Either achieved by a Software or Combined Attack, a type confusion can have various origins. The following gives a classification for these origins and an

[1] The .CAP file for Converted APplet, obtained from the standard .CLASS file generated by the Java compiler.

example of a practical attack for each class. The two first classes concern software attacks based on ill-formed or ill-verified applications which imply that the attacker can load any application on the platform. The three other classes consider a combination of software and fault injection attacks that only require that the attacker can load verified applications. Finally, this section will present another classification for the attacks which may appear less specific but will better suit the remainder of our analysis.

Software Attack: Ill-Formed Application (SA-IFA). In the case one is able to load any application onto the card without executing the BCV, one can freely load a corrupted binary file representing an ill-formed application that does not comply to the Java Card language rules.

In particular, it is well known that it is not possible to manipulate directly the length of an array of byte after its instantiation in Java. A typical attack would then consist in loading the corrupted binary file representing the code depicted in List. 1.1 as proposed in [2].

Listing 1.1. Example of an SA-IFA

```
public class Fake { short len = (short) 0x7FFF; }

byte[] bArray = new byte[10];
Fake f = bArray;
// Subsequent manipulations of f.len allow to modify bArray's
// length
```

By modifying the length of the array, one may be able to access data out of its bounds and then to get a dump of the card.

Another classical example of an SA-IFA consists in setting the reference of a given object through an integer value (List. 1.2), which is prohibited by the language and leads to an error during the verification process. Indeed, modification of the .CAP file is mandatory here since any Java compiler would raise an error on such Java sources.

Software Attack: Ill-Verified Application (SA-IVA). Another way of having an ill-typed application loaded into the card is to use a particular library already on-card and to feed the BCV with an erroneous export file for this library[2]. Again, this would allow one to escape the language rules and in particular those concerning type safety.

An example of such an attack abusing the sharing mechanism (derived from that described in [2]) is presented in List. 1.3. Using the corrupted export file, the BCV will pass and the byte array sent by the Client will be used as a short array by the Server.

[2] This file will be used as the definition of the library by the off-card verification process.

Listing 1.2. Example of an SA-IFA (2)

```
// Receive reference through the APDU buffer
int reference = getValue(apduBuffer);

// Assign it to the object instance
myObject = reference;
// The above line can be obtained for instance by compiling:
// reference = reference;
// myObject = myObject;
// And replacing the unwanted bytecodes by NOPs in the CAP
```

Listing 1.3. Example of an SA-IVA

```
//  Server CAP file: Server uses short[]
public interface MyInterface extends Shareable {
    void accessArray(short[] array); }

//  Server EXP file: Server assumes byte[]
public interface MyInterface extends Shareable {
    void accessArray(byte[] array); }

//  Client CAP file: Client assumes byte[]
public interface MyInterface extends Shareable {
    void accessArray(byte[] array); }
```

By accessing a byte array as a short array, one may be able to access twice as much data as it should be allowed to, getting here again a partial dump of the card.

Combined Attack: Data Disruption (CA-DD). The different ways to combine software and fault attacks mainly differ in the effect of the fault injection. Firstly, we consider that the attacker can modify the data manipulated by the application.

Such an attack can for example consist in creating numerous instances of a given class B and attempt to corrupt the reference of an instance of another class A so that it actually points to one instance of class B. This attack was firstly described on standard Java platforms in [31] and adapted on Java Card platforms in [32]. List. 1.4 recalls it briefly. By modifying the reference of *b.a1* and by manipulating the short value *b.a1.s1*, one can access an instance field she should not.

Combined Attack: JC Application Code Disruption (CA-JCACD). It is also possible for an attacker with fault injection capabilities to target directly the code of her own Java Card application. This option was for instance used

Listing 1.4. Example of a CA-DD

```
public class A { short s1, s2, s3, s4; }
public class B { A a1, a2, a3, a4; }

// Create as many instances of class B as possible
B b0000 = new B(); B b0001 = new B(); ... B b00FF = new B(); ...

// Create one instance of class A
A a = new A();

// A perturbation of the reference of a will most likely make it
// point to an instance of class B
a.s1++;      // actually increments the reference of b.a1
```

in [23] to return a value that will be used as the reference of a *Key* object by forcing a bytecode instruction to 0, corresponding to the bytecode *NOP* (for No OPeration), as recalled in List. 1.5.

Listing 1.5. Example of a CA-JCACD

```
Key getKey(short index)
{
  // The if-statement is coded as: 1D 1077 6D08
  // Forcing to 0 the instruction bspush (10) results in coding
  // the following statement:
  //    return index;
  if (index<0x77)
    return keys[index];
  else
    return null;
}
```

In another kind of application code disruption, we can also refer to the work by Bouffard *et al.* [25] which shows that the perturbation of a branch instruction (a *goto_w* at the end of a for-loop to be specific) can lead to the execution of arbitrary instructions stored in a static array of byte.

Combined Attack: JCVM Code Disruption (CA-JCVMCD). Another kind of disruption that can lead to a type confusion is the perturbation of the execution of the JCVM itself.

An example of such an attack was presented in [24] where the *checkcast* instruction, which is meant to verify dynamically the validity of explicit type

casting, was disrupted to provoke a type confusion. The authors exploit then this fault to take advantage of the specific features of the Java Card 3 *Connected* Edition platform, yet it can totally be used on *Classic* Editions as shown in List. 1.6.

Listing 1.6. Example of a CA-JCVMCD

```
public class A { C c; }
public class B { short s;}

A a = new A();
// Explicit casting to B forces the compiler to generate a
// checkcast instruction which can be disrupted
B b = (B) (Object) a;
// Subsequent manipulation of b.s allow to modify c's reference
b.s = sRef;
```

Table 1 sums up the different attack classes and states the effectiveness of the only countermeasure initially present to counteract these attacks. All along this article we will enrich this table in order to assess previous works as well as the propositions presented in the remainder.

Table 1. Efficiency of the ByteCode Verifier versus the different classes of attack leading to type confusion

	SA-IFA	SA-IVA	CA-DD	CA-JCACD	CA-JCVMCD
ByteCode Verifier	Yes	Yes, with correct .EXP files	No	No	No

Furthermore, beyond this classification we also denote that two different kinds of type confusion can occur on the platform:

- Direct type confusion : As in List. 1.5, direct type confusion consists in directly using an object reference as an integer value (or *vice versa*).
- Indirect type confusion : As in List. 1.4 and 1.6, indirect type confusion consists in using an object reference as an integer value (or *vice versa*) through two object instance fields.

One should also note that except the confusions involving arrays, all attacks are based on an integer-to-Object type confusion which allows to manipulate object references. We will pay a particular interest to this specific kind of type confusion in following sections.

3 A Survey of Previous Works

This section gives a brief overview of different solutions enabling to counteract some type confusion attacks presented in the literature. As previously stated, we focus in the remainder on type confusions between integer values and object references. For each of the presented work, we refer the reader to the original articles for a detailed description.

3.1 Redundant Stack Checks

Description. In [27] Barbu *et al.* introduce an element within the Java Card runtime frame of execution allowing them to check the integrity of the data read and written (i.e. pushed onto and popped from) the operand stack. Indeed, given that any data pushed onto the operand stack is meant to be popped from the operand stack at a certain point, they exhibit an invariant relation permitting to somehow check the operand stack integrity.

Analysis. The proposed countermeasure actually procures a protection against type confusion based on operand disruption. However this method fails to detect an error if the operand stack is not directly targeted, which limits its overall effectiveness.

3.2 The Typed Stack

Description. In [33], Dubreuil *et al.* proposed to enhance the robustness of Java Card platforms against type confusions thanks to the use of a typed operand stack. Their proposal consists in dynamically splitting the operand stack into two parts dedicated to integer values on the one hand and to object references on the other hand as depicted in Figure 1.

Analysis. As claimed by the authors, this countermeasure is very efficient in detecting direct type abuse such as the one described in List. 1.5 where an integer value is returned as a reference to a *Key* object. Indeed, since the integer would be pushed on one side of the stack, trying to pop a reference from the other side of the stack would result in a stack underflow error. It is also worth noticing that the cost for this countermeasure at the JCVM level is really negligible since only an additional pointer is required to keep track of both the integer-top-of-stack and the reference-top-of-stack.

However, the authors also claim that the typed stack allows to counteract attacks like that presented in Section 2, based on an indirect type confusion. This turns out to be incorrect since in this case the confusion occurs within an object instance field, i.e. away from both sides of the operand stack. Besides, not only this attack is not detected by the typed stack but it also allows one to circumvent the security brought by the typed stack. Indeed, once an indirect type confusion has been achieved, executing any type confusion just require an additional step through the involved instance fields to avoid the check enforced by the typed stack without any additional fault injection.

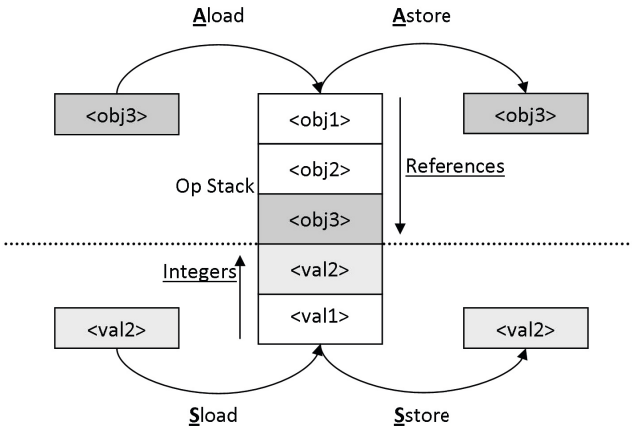

Fig. 1. The typed stack

3.3 Integrity Checks

Description. In [32], Lancia proposed to add integrity checks within object instances in order to detect possible disruptions of instance fields and thus to prevent type confusions due to data disruptions (CA-DD).

Analysis. Obviously, such a countermeasure is only meant to avoid CA-DD attacks and will not detect any of the other attacks previously described. Furthermore, the cost of this mechanism both in terms of memory footprint and execution time should not be neglected.

3.4 Typewise Masking

Description. Further proving the industrial interest in type confusion attacks, Girard *et. al* filed a patent [34] in which they propose a typewise masking scheme randomizing the effects of a successful type confusion. Their idea consists in applying a transformation to data manipulated in RAM (i.e. local variables and instruction operands) depending on their primitive type by associating one different random number to each type: *integer*, *boolean*, *character*, *reference*, other (*double*, ...). In order to be able to execute operations on these data, the mentioned transformations need to come with their reciprocals, such as Exclusive OR/Exclusive OR and Addition/Substraction for instance. The authors argue that succeeding in a type confusion, an attacker would only be able of manipulating random variables and would consequently not threaten the security of the card.

Analysis. If such a technique is actually efficient at countering the attack described in List. 1.2 (the value stored in *myObject* would indeed be random), a small modification of the malicious code is sufficient to render it useless. For instance, the boolean masking can be circumvented by executing the sequence

`myObject = reference ^ null;` and the additive masking by the sequence `myObject = reference + null;`. Furthermore, an attack such as presented in List. 1.4 would not be thwarted. We detail the attacks on [34] in Appendix A.

3.5 Limitations of State-of-the-Art Countermeasures

Table 2 sums up the efficiency of the different countermeasures presented so far with regards to the attacks introduced in Section 2. Analysing Tab. 1, one can

Table 2. Efficiency of published countermeasures versus the different classes of attack leading to type confusion

	SA-IFA	SA-IVA	CA-DD	CA-JCACD	CA-JCVMCD
ByteCode Verifier [6]	Yes	Yes, with correct .EXP files	No	No	No
Stack Invariant [27]	No	No	Yes	No	No
Typed Stack [33]	Yes	Yes	No	Partial	Partial
Field Integrity [32]	No	No	Partial	No	No
Typewise Masking [34]	No	No	No	No	No

immediately see that none of the solutions proposed so far can withstand all the identified attack scenarios. Furthermore, we can see that even a combination of different countermeasures offers only a partial coverage of the attack paths.

Indeed Tab. 3, showing the behaviours of the state-of-the-art methods with regards to whether the type confusion is direct (DTC) or indirect (ITC), better reflects the limitations of the proposed countermeasures.

Again, it is obvious that none of the state-of-the-art countermeasures fully cover the identified threats, and even combining the different countermeasures, one does not reach a complete security, particularly because of indirect-type-confusion-based Combined Attacks. Section 4 introduces two embodiments of a platform-wide approach to prevent both direct and indirect type confusion.

4 A Platform-Wide Approach to Type Confusion Detection

As we have seen, the main drawback of the different countermeasures presented so far is that they focus on one particular part of the JCRE. Therefore each solution fails to detect the type confusion or its exploitation as soon as it occurs out of the scope of the protected area. In order to detect such flaws, one needs to propagate the type information within the whole runtime environment.

Table 3. Efficiency of published countermeasures versus the different classes of attack leading to type confusion

	SA-DTC	SA-ITC	CA-DTC	CA-ITC
ByteCode Verifier [6]	Yes	Yes	No	No
Stack Invariant [27]	No	No	Partial	No
Typed Stack [33]	Yes	No	Yes	No
Field Integrity [32]	No	No	No	Partial
Typewise Masking [34]	No	No	No	No

4.1 Random Transformation of Object References

In this section we intend to propose a very memory-efficient method that does not aim at detecting the type confusion itself but rather at preventing an attacker from using integer-to-Object confusion with an overwhelming probability.

Everything Is NOT an Object. As highlighted in Section 2, a particularly attractive capacity endowed by an integer-to-Object type confusion is to use a kind of C-like pointer arithmetic on object references. It is indeed generally easy for an attacker to perform such operations since object references are usually affected linearly, cf. [32]. Therefore once one reference is obtained through the initial type confusion, she can increment/decrement it to access potentially all other object instances.

The base idea of our proposal is that the number of object instantiated on a platform is rather small in practice. Assuming object references are coded on N-bit words, we argue that only a small fraction of these values are actually used. We then have a large number of references that do not point to any object instance.

Definition 1. *Let $\mathcal{M}$ be the set of values returned by the implementation of the bytecode instruction* new *for all created object instances and $\mathcal{N}$ be the set of all possible N-bit reference values.*

According to Def. 1, all values in $\mathcal{N} \backslash \mathcal{M}$ are undefined references. Given that these undefined references should never be used by the JCRE, we can easily imagine that a strong security action is taken whenever an application attempts to access it. The security of our scheme relies on this last statement that the platform does not allow an attacker to repeat access attempts endlessly (i.e. until she meets success) as well as on the order of magnitude between the size of the two sets $\mathcal{N}$ and $\mathcal{M}$.

The Random Transformation Scheme. In order to prevent the attacker from gaining access on the reference allocation mechanism and more precisely on the values of the references, we propose to define a secret random transformation scheme to randomly inject the values of $\mathcal{M}$ in $\mathcal{N}$, such a transformation being specific for each card. Therefore, the probability that the attacker gains access to an unknown reference will be relative to the ratio $\frac{\#\mathcal{M}}{\#\mathcal{N}}$.

The following describes how this can easily be achieved by applying a random affine function to each and every reference[3].

Definition 2. *Let $f : X \to a \cdot X + b \bmod p$, with p the largest prime number such that $p \leq 2^N - 1$ and a and b two random numbers drawn in $[1, p[$.*

Property 1. One should note that f is a bijective function from $\mathbb{F}_p$ to $\mathbb{F}_p$ since $gcd(a, p) = 1$ ensures that $\forall\ a \in \mathrm{GF}(p)^*,\ \exists\ i_a = a^{-1} \bmod p$ such that $f^{-1}(Y) = (Y - b) \cdot i_a \bmod p$.

The following shows that it is possible to use such a function f to randomise object references while preserving the compatibility with all instructions of the standard Java Card instruction set.

Proposition 1. *If*

- *for all created object instance with reference X, the JCRE transforms X into $Y = f(X)$, and*
- *for all manipulated integer value S, the JCRE transforms S into $T = S \oplus c$ (with c a random number drawn in $[1, p[$),*

then every Java Card standard instruction can be executed by reversing the transformation.

Proof. Each instruction of the instruction set manipulates either

a reference: *in this case the JCRE can use $X = (Y - b) \cdot a^{-1} \bmod p$;*
an integer: *in this case the JCRE can use $S = T \oplus c$;*
an untyped: *in this case the JCRE does not need to evaluate the data being manipulated since dup* and pop* instructions only duplicate or discard operands regardless of their type.*
no data: *obviously in this case nothing needs to be done.*

□

Proposition 2. *If*

- *for all created object instance with reference X, the JCRE transforms X into $Y = f(X)$, and*
- *for all manipulated integer value S, the JCRE transforms S into $T = S \oplus c$,*

then the probability P that an attacker setting the reference of an object to an arbitrary value actually sets a valid object reference is as low as:

$$P = \frac{\#\mathcal{M}}{p - 1} \tag{1}$$

[3] Note however that any bijective function $f : \mathcal{N} \to \mathcal{N}$ would suit our purpose.

even in the case where the attacker is able to read a stored reference through an integer local variable or instance field.

Proof. Let S be the integer value set by the attacker. The JCRE will actually store $T = S \oplus c$, with c a random value unknown from the attacker. Therefore even in the case where S is known to be the image by f of a valid reference, the stored integer T can be considered as a random variable.

Now given a random value $T \in [0, p[$, the probability P that it corresponds to one of the $\#\mathcal{M}$ images of function f applied on an existing reference can be straightforwardly computed to obtain the stated result. □

We assume that short values are used in practice. Therefore, we let $N = 16$ and $p = 65\,521$ in the rest of this section.

Analysis. According to this scheme, an attacker attempting to set the reference of an object would trigger a security action with a certain probability depending on the number of valid references the platform supports. Fig. 2 gives an estimation of the number of valid references depending on the memory available for Java objects and the average size of an object instance on the platform.

We depict in Fig. 3 the success rate of an integer-to-Object confusion depending on the number of valid references. As we can see, an attacker attempting to set the reference of an object would trigger a security action with an overwhelming probability in most practical cases. For instance, the attack detection probability reaches 96.9% if we consider 2 048 objects simultaneously instantiated. Provided this security action is sufficiently severe, we can then conclude that the security level ensured by this proposition is satisfying, although only theoretically probabilistic. However, it is only fair to recall that this method detects integer-to-Object type confusions but only randomizes the other-way-round confusion. Yet this latter appears much less concerning from a security point of view.

Now considering the memory footprint penalty, it is excellent since it requires 5 words to be stored only: the three random parameters a, b and c, the value $a^{-1} \bmod p$ and the prime p. Regarding the time penalty, executing the affine function costs one multiplication, one addition and one modular reduction on N-bit words, that is to say 5 clock cycles per f evaluation on components using an efficient assembly language.

Finally, it can be mentioned that unlike the proposal of Girard *et al.* presented in Section 3.4, our scheme does not use the same transformation for integers and references, which prevent it from suffering the same flaws.

4.2 The JCVM Abstract Companion

Section 4.1 has shown an efficient probabilistic way to reflect the type of manipulated data. In a second embodiment of the global approach for type safety, we propose to run together with the JCVM what we call an *Abstract Companion*. The aim of this JCVM Abstract Companion is to duplicate the behaviour of the

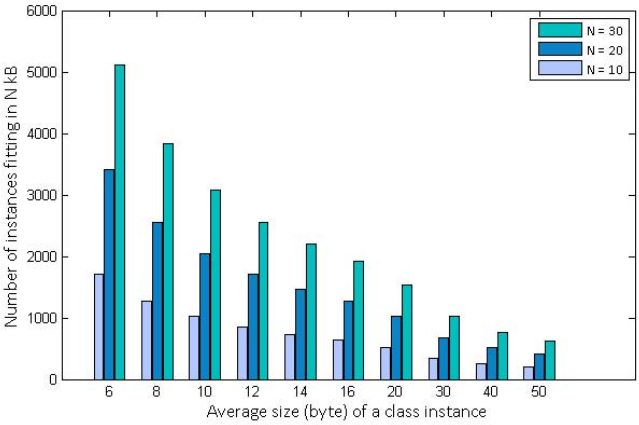

Fig. 2. Number of valid reference depending on available memory and average object instance size on the platform

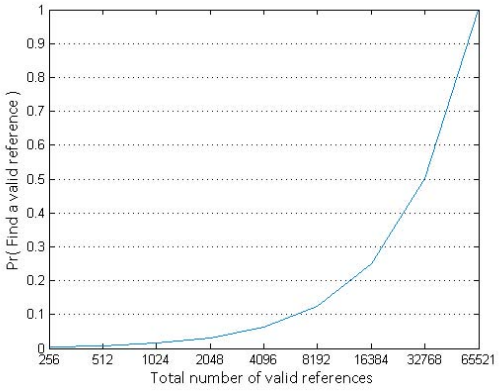

Fig. 3. Success rate of a confusion Vs Number of valid reference on the platform

JCVM but manipulating only boolean values reflecting the type (i.e. whether it is a reference or an integer) of the data simultaneously manipulated by the JCVM, hence the *Abstract* qualifier.

In order to implement this companion, one only needs to take care of three different structures: the operand stack, the local variables table and the instance fields. The modification to apply to the JCVM are detailed hereafter.

The Abstract Operand Stack. The operand stack is the central element of the JCRE. Indeed it is the data structure where all parameters and returned values of the bytecode instructions are pushed-on and popped-out. Concerning the operand stack, the most straightforward and efficient technique to keep track of

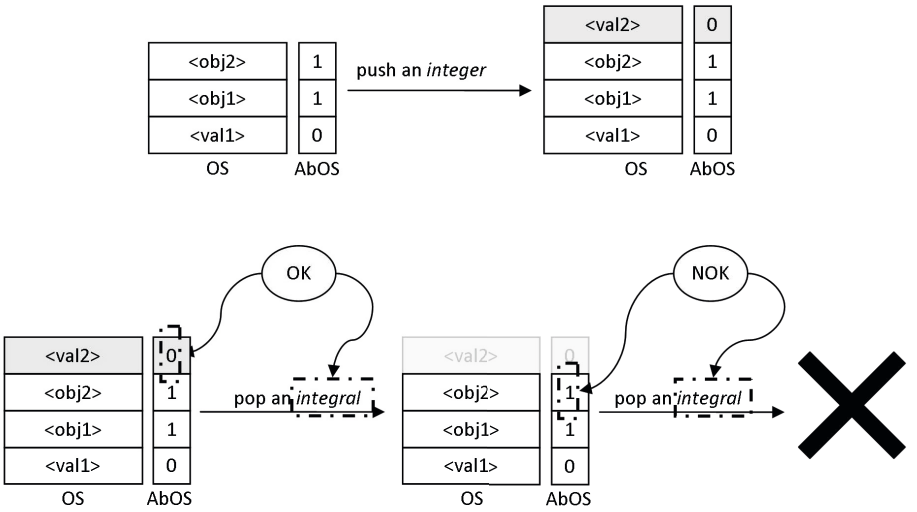

Fig. 4. The abstract stack

the type of the top-of-stack value at any time with a minimal modification of both the JCVM and the applets to be loaded consists in implementing an abstract operand stack where a single bit (0 or 1) reflects the type (integer or reference respectively) of the corresponding data in the operand stack. Consequently, all instructions implemented by the JCVM need to repeat on the abstract stack their action on the regular stack.

For instance, if we consider the instruction *aload* which pushes onto the stack a local variable of type reference, this instruction will also have to set the top-of-abstract-stack bit to 1. Then it is straightforward to see that an operation that is not consistent with the type of an operand will be detected by the abstract operand stack. For instance, if the top-of-abstract-stack bit is set to 1, any instruction operating on a data of type integer will be rejected. Fig. 4 illustrates the principle of the abstract operand stack.

It is worth noticing that the abstract operand stack is fully compliant with all standard instructions, unlike the typed stack proposed in [33] which does not support untyped instructions: *dup**, *pop**. Yet this comes at the price of a superior cost in memory footprint. Our method requires $2^{\lceil log(max\_stack\_size) \rceil}/2^8$ bytes (considering the general case where memory is byte-addressable) whereas the method of Dubreuil *et al.* only requires one additional word (say two bytes) but comes at the price of rewriting all incompatible operations in each and every Java Card applet loaded on the platform.

The Abstract Local Variable Table. Regarding the local variable table, a very similar approach can be used. At first, one can think of assigning each cell a particular type, for instance at the first access to this cell. However, the fact that a single cell of the table may be used to store different local variables in the framework of a single method execution forces us to be less restrictive.

We therefore propose to define a bitwise abstract local variable table that is updated each time a local variable is stored and checked against the type of the bytecode instruction whenever a variable is loaded. For instance, executing the bytecode instruction *sstore_1* will reset bit 1 of the table to 0 and if the bytecode instruction *aload_1* is subsequently executed then an error would be raised since the value 1 would be expected at bit 1 of the abstract local variable table.

Abstract Trusted Instances. Finally, the type of the fields of all objects instantiated on the platform need to be verified. For that purpose, we suggest to define what we call an abstract trusted instance for each class loaded on the platform. This trusted instance can be store in Non-Volatile Memory together with the class definition. Its structure shall be identical to that of future instance of this class, except that each field is represented with one bit set either to 0 or 1 depending on whether the field is of type integer or reference.

As for the previous structure, at runtime, any access to the field of a given class instance should be checked against this class trusted abstract instance and execution should continue only if this check deems successful.

Analysis. The JCVM Abstract Companion is no more than a way of keeping track of the type information all along the execution of an application. Its representation is compact as can be since we use only the bit of information that is necessary to hold the boolean statement saying whether a data is of type reference or not (save the possible space lost due to byte-addressable memory). Furthermore, it appears suitable for any JCVM since it does not rely on any specific implementation choices and we believe that it should not require major modifications for any JCVM's implementation to support it. However, we add to each of the described structure an area of size of the order of $log(S_i)$ bytes,

Table 4. The different classes of attack leading to type confusion

	SA-DTC	SA-ITC	CA-DTC	CA-ITC
ByteCode Verifier [6]	Yes	Yes	No	No
Stack Invariant [27]	No	No	Partial	No
Typed Stack [33]	Yes	No	Yes	No
Field Integrity [32]	No	No	No	Partial
Typewise Masking [34]	No	No	No	No
Random Transformation [§4.1]	Yes with $P \sim 97\%$	Yes with $P \sim 97\%$	Yes with $P \sim 97\%$	Yes with $P \sim 97\%$
Abstract Companion [§4.2]	Yes	Yes	Yes	Yes

with S_i the size of these structures. The cost in terms of memory footprint of this solution is therefore non-negligible.

4.3 Summary

Tab. 4 sums up the results obtained with the proposed method and compares them to the countermeasures presented in Section 3. As we can see both our proposals achieve a better coverage regarding direct and indirect type confusion from integer to reference.

5 Conclusion

In this article, we have pointed out through a comprehensive analysis of the state-of-the-art that several attack paths involving type confusion presented in the literature have never been fully covered. This lack is mainly due to the fact that all proposed countermeasures mainly focused on one specific area, either the operand stack or instance fields. Starting from this point, we have shown that a generic platform-wide approach is possible and we have proposed two particular schemes implementing this approach. The first one, based on a card-specific random transformation, provides a probabilistic security against integer to reference type confusion (also across the whole platform). Nevertheless, it was shown that the probability of success for an attacker is extremely low and that this technique can be implemented at a very low cost. On the other hand, the JCVM Abstract Companion has been shown as a resource-demanding but very effective security mechanism since it is meant to detect each and every type confusion between integers and Objects across the whole platform.

References

1. Witteman, M.: Java Card Security. Information Security Bulletin 8, 291–298 (2003)
2. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
3. Séré, A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic Detection of Fault Attack and Countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security, WESS 2009, pp. 1–7 (2009)
4. Hogenboom, J., Mostowski, W.: Full Memory Attack on a Java Card. In: 4th Benelux Workshop on Information and System Security, Louvain-la-Neuve, Belgium (2009)
5. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan Applet in a Smart Card. Journale on Computers and Virology 6, 343–351 (2010)
6. Lindholm, T., Yellin, F.: Java Virtual Machine Spec. 2nd edn. Addison-Wesley, Inc. (1999)
7. Oracle Corp.: Virtual Machine Spec. – Java Card Plateform, Version 3.0.4 Classic Ed. (2011)

8. Oracle Corp.: Application Programming Interface, Java Card Platform, Version 3.0.4 Classic Ed (2011)
9. Oracle Corp.: Runtime Environment Spec. Java Card Platform, Version 3.0.4 Classic Ed. (2011)
10. Sun Microsystems Inc.: Virtual Machine Spec. – Java Card Plateform, Version 3.0.1 Connected Ed. (2009)
11. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform, Version 3.0.1 Connected Ed. (2009)
12. Sun Microsystems Inc.: Java Servlet Spec., Java Card Platform, Version 3.0.1 Connected Ed. (2009)
13. Sun Microsystems Inc.: Runtime Environment Spec., Java Card Platform, Version 3.0.1 Connected Ed. (2009)
14. Barbu, G.: Fault Attacks on Java Card 3 Virtual Machine. In: e-Smart 2009 (2009)
15. Bellcore: New Threat Model Breaks Crypto Codes. Press Release (1996)
16. Boneh, D., De Millo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
17. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
18. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Applications VI – CARDIS 2004, pp. 159–176. Kluwer Academic Publishers (2004)
19. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. IEEE 94, 370–382 (2006)
20. Vertanen, O.: Java Type Confusion and Fault Attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 237–251. Springer, Heidelberg (2006)
21. Skorobogatov, S., Anderson, R.: Optical Fault Induction Attack. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
22. Quisquater, J.J., Samyde, D.: Eddy Current for Magnetic Analysis with Active Sensor. In: e-Smart 2002 (2002)
23. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
24. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
25. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
26. Barbu, G., Thiebeauld, H.: Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 18–33. Springer, Heidelberg (2011)
27. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg (2011)

28. Bouffard, G., Lanet, J.-L.: The Next Smart Card Nightmare, Logical Attacks, Combined Attacks, Mutant Applications and Other Funny Things. In: Naccache, D. (ed.) Quisquater Festschrift. LNCS, vol. 6805, pp. 405–424. Springer, Heidelberg (2012)
29. Barbu, G., Hoogvorst, P., Duc, G.: Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS 2012. LNCS, vol. 7159, pp. 1–13. Springer, Heidelberg (2012)
30. Barbu, G.: On the Security of Java Card Platforms against Hardware Attacks. PhD thesis, Télécom ParisTech – Institut Télécom (2012)
31. Govindavajhala, S., Appel, A.: Using Memory Errors to Attack a Virtual Machine. In: IEEE Symposium on Security and Privacy, pp. 154–165. IEEE Computer Society (2003)
32. Lancia, J.: Java Card Combined Attacks with Localization-Agnostic Fault Injection. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 31–45. Springer, Heidelberg (2013)
33. Dubreuil, J., Bouffard, G., Lanet, J.L., Cartigny, J.: Type classification against fault enabled mutant in java based smart card. In: ARES, pp. 551–556. IEEE Computer Society (2012)
34. Girard, P., Gonzalvo, B.: Making Secure Downloaded Application in Particular in a Smart Card. Publication Number: FR2266222 - US7168625, Gemplus (2003)

A Attacking Typewise Masking

In Section 3.4, we claim that the typewise masking can be easily circumvented by a slight modification of the malicious application involved. In the following we detail the execution of the initial and modified malicious line of code for both the additive and boolean masking schemes, together with the evolution of the content of the operand stack. We refer to the object type and integer type masks as M_A and M_I respectively.

Additive Masking.

Initial malicious code: `myObject = reference;`

`Executed Bytecode`	*Operand Stack*
`iload <reference>`	$[reference + M_I]$
`astore <myObject>`	$[-]$

Consequently, the value stored in `myObject` is actually set to the random value: $reference + M_I - M_A$ and the attack is either thwarted or detected.

Modified malicious code: `myObject = reference + null;`

`Executed Bytecode`	*Operand Stack*
`aconst_null`	$[0 + M_A]$
`iload <reference>`	$[M_A, reference + M_I]$
`iadd`	$[((M_A - M_I) + ((reference + M_I) - M_I)) + M_I]$ $[M_A + reference]$
`astore <myObject>`	$[-]$

Consequently, the value stored in `myObject` is actually set to the targeted value: $reference$ and the attacker can carry on.

Boolean Masking.

Initial malicious code: `myObject = reference;`

Executed Bytecode	*Operand Stack*
`iload <reference>`	$[reference \oplus M_I]$
`astore <myObject>`	$[-]$

Consequently, the value stored in `myObject` is actually set to the random value: $reference \oplus M_I \oplus M_A$ and the attack is either thwarted or detected.

Modified malicious code: `myObject = reference ^ null;`

Executed Bytecode	*Operand Stack*
`aconst_null`	$[0 \oplus M_A]$
`iload <reference>`	$[M_A, reference \oplus M_I]$
`ixor`	$[((M_A \oplus M_I) \oplus ((reference \oplus M_I) \oplus M_I)) \oplus M_I]$ $[M_A \oplus reference]$
`astore <myObject>`	$[-]$

Again, the value stored in `myObject` is actually set to the targeted value: $reference$ and the attacker can carry on.