Java Card Combined Attacks with Localization-Agnostic Fault Injection

Julien Lancia

SERMA Technologies, CESTI, 30, avenue Gustave Eiffel
33600 Pessac, France
j.lancia@serma.com

Abstract. In this paper, we present a paradigm for combined attacks on Java Cards that lowers the requirements on the localization precision of the fault injection. The attack relies on educated objects allocation to create favorable memory patterns that raise the chances of success of the combined attack. In order to maximize the probability of successful injection, we determine the optimal parameters depending on the physical properties of the targeted platform. Finally, we demonstrate the efficiency of our approach through fault injection simulation.

1 Introduction

Security in the smart card field is a main issue, due to the very nature of the data stored on chips. Up to now, several attack paths have been explored to compromise the security of these platforms [6,10,13,16,1,11], leading to new counter measures designed to prevent these attacks from succeeding [17,5,19,12]. In order to strengthen the security of the embedded systems, the Java Card approach is to rely on a strong-typed language and to factorize the security verifications in the Java Card Virtual Machine (JCVM) abstraction layer. The JCVM provides several security services to the application layer: the firewall mechanism enforces applets isolation, the JCVM checks for stack- and buffer-overflow in order to prevent illegal memory accesses, and finally the bytecode verifier (BCV) is responsible for guaranteeing the type safety of the executed programs.

Up to the 2.2.2 version of the Java Card specifications [23,24,22,25], the bytecode verification is performed off-card, and is therefore optional. Many attacks on the Java Card platforms [18,9,8] are based on manipulations of the binary representation of the applet (the cap file) aiming a circumvention of the type system of the virtual machine. Such attacks, called "logical attacks" have indeed huge impacts on the overall security of the platform, but are irrelevant on industrial processes where bytecode verification is mandatory. Moreover, in the last version of the Java Card specifications (3.0 connected edition [26,27,28]), the bytecode verifier is embedded on card, making the bytecode verification process mandatory.

Recently, a new paradigm of attacks called "combined attacks" emerged [3,29,4]. These attacks combine logical attacks and physical attacks to bypass the protection mechanisms of the Java Card platform. Physical attacks on smart cards usually rely on laser beams[20], which have to be precisely tuned both in timing and localization for the combined attack to succeed. Therefore time and geographic localizations are strong constraints on the success rate of the combined attacks.

In this paper, we propose a paradigm for combined attacks on smart cards that allows an attacker to evade localization constraints of the hardware attack. This paradigm, inspired from an attack aiming classical Java Virtual Machines (JVM) [15], is based on specific memory patterns that are obtained through objects allocations by a malicious applet. These objects allocations, although legal regarding the Java Card specifications and bytecode verifier, end up with a particular memory pattern that raises the chances of success of the fault injection, regardless of the localization of the laser pulse.

The rest of this paper is structured as follows. In Section 2, we briefly present the concepts of the original attack on a JVM, we explain why this attack can't succeed on modern JVCM implementations, and we expose our work to exploit the original concept as part of a combined Java Card attack. In Section 3, we present our experimental results. In Section 4, we discuss how to prevent such attacks. In Section 5 we compare our work to existing approaches and conclude on the contribution this work brings to the Java Card attacks field.

2 The Attack Concept

2.1 Exploitation of Memory Errors on a Java Virtual Machine

Although using physical fault injection on a system's memory to evade software security mechanisms has been practiced for a while now on smart cards, the idea to induce memory errors likely in a PC's SRAM through physical fault injection has emerged recently. In [7], Govindavajhala and Appel present a concept of attack that allows arbitrary memory error to produce type confusion in a classical Virtual Machine.

Their attack applet declares the two classes A and B presented in Figure 1. The applet fills the heap with many instances of class *B* and one instance of class *A* named *a*. All the fields of all the *B* instances are initialized to point to the unique *A* instance. Figure 2 represents an excerpt of the memory after object initialization.

The algorithm of the applet loops through all *A* fields of all the objects in memory and checks through Java pointer equality whether they still contain the address of the A instance (noted *x*). In case an error switches any *A* field's value (let's say b31.a2), the original algorithm of the applet presented in Listing1.1 mutates into a two steps type confusion on the erroneous field reference.

Listing 1.1. Two steps type confusion on an erroneous field reference

```
A aObj; B b31; B fakeB;
aObj = b31.a2;      // type confusion, step 1
fakeB = aObj.b;     // type confusion, step 2
```

```
class A {                       class B {
  A a1;                           A a1;
  B b;                            A a2;
  int addr;                       A a3;
  A a4;                           A a4;
};                              };
```

Fig. 1. Classes of the original applet

Figure 2 illustrates the memory operations performed by the virtual machine during these two steps, with and without a fault injection. In step 1, the applet dereferences the b31.a2 field in aObj. The memory error changes the content of the aObj field, that normally contains the address x of the A instance, into a new value $x'=x+\Delta x$, . In step 2, the applet dereferences the B field aObj.b. Because the memory error induced a Δx bias of the aObj value, the resulting reference is likely to contain an A reference as most of the memory is filled with fields of type A. This produces a type confusion as the *fakeB* reference, that is statically typed as a B object, references an object of type A.

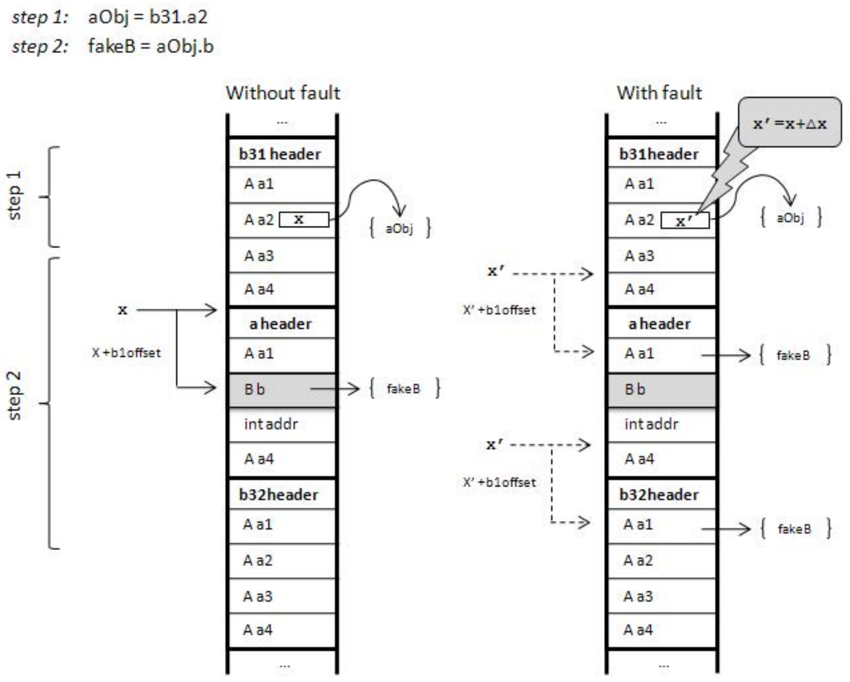

Fig. 2. Realization of the type confusion on a Java Virtual Machine

The exploitation of this confusion is very straightforward: the *addr* field of the unique *A* instance is set to forge an *A* object reference at the address *custom_ addr*ess, accessible through fakeB.a3 (as *addr* is the third field of *A*). The field fakeB.a3.addr sits in memory at the address *custom_ address+a3offset*. All the addressable memory is therefore accessible for reading and writing through this field, with a constant memory offset equal to *a3offset*. The exploitation of the type confusion is synthesized in Figure 3.

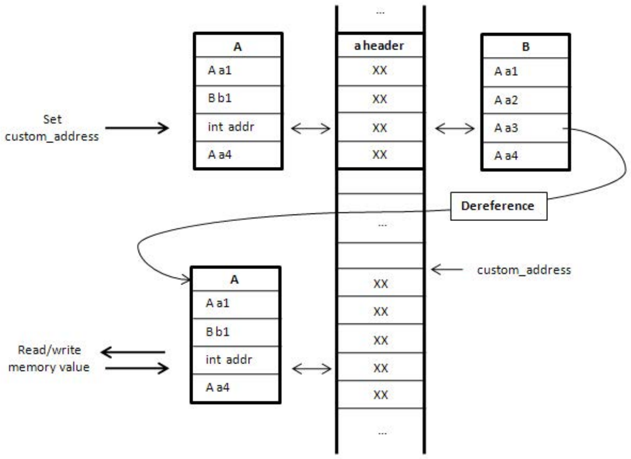

Fig. 3. Exploitation of the type confusion on a Java Virtual Machine

2.2 Specificity of Modern Java Card Virtual Machines

The attack presented in the previous subsection is dedicated to classical Java virtual machines. It could not be led identically on Java Card virtual machines because the embedded platforms they run on have very specific hardware constraints, that impacts the memory layout of the virtual machine instances and the physical properties of the memory where these instances are stored.

More precisely, Java virtual machines run on standard computer architectures, where object instances are allocated in RAM memory. Contrarily, in Java Card virtual machines, instances are allocated in persistent memory (i.e. flash or EEPROM).[1]

The physical properties of the persistent memories make it much more difficult for an attacker to realize a bit flip through external means [21]. In opposite to

[1] Java Card platform provides methods to create temporary (transient) arrays whose content is stored in RAM, but their headers are still in persistent memory. Only the objects stored in transient object arrays would have their headers in RAM memory. These objects are mandatorily of type Object, and are consequently not subject to type confusion.

the attack presented in the previous section where the faults in RAM memory are triggered using the heat of a portable light, errors in persistent memory generally require laser or electromagnetic injections [2,20]. Moreover, because of the scarcity of the resources in these embedded platforms, the amount of persistent memory available for object instantiation is quite limited. The memory layout of the objects used for the attack must therefore be optimized in order to maximize the chances of success.

Another main difference between standard Java virtual machines and Java Card virtual machines is the memory management. In standard Java virtual machines, references are represented using a direct memory addressing, which means that the physical memory address of the referenced instances are stored in memory like traditional pointers. In modern implementations of Java Card virtual machines, references are represented using an indirect memory addressing, where references are represented as an index in a global instance pool managed by the virtual machine. This difference between the Java and Java Card virtual machines addressing mode is illustrated in Figure 4. Because the indirect memory representation of the instances is decorrelated from the physical address of the instance, the attack presented in previous section can not apply on Java Card virtual machines and must therefore be adapted for these platforms.

The use of indirect addressing mode in Java Card virtual machine has another impact that concerns the exploitation of type confusion. A type confusion between a short field and an instance field in a direct memory addressing mode can easily be exploited by using the value of the short field as a pointer address to scan the whole memory. In case of an indirect memory addressing mode, the same type confusion gives access to an index in the global instance pool of the virtual machine. Therefore, the exploitation of the type confusion to scan the memory requires several additional operations. These operations are detailed further in Section 2.3.

2.3 Combined Attack on a Java Card Virtual Machine

Realization of the Attack. Combined attacks aim at taking benefit from hardware and logical attacks to create applets that are considered legitimate by the Java Card virtual machine and the verifier, and whose attack load is activated through fault injection. Our attack maximizes the chances of success of the fault injection by creating favorable pattern in persistent memory through instances allocation. As a result, almost any bit switch in persistent memory creates an exploitable type confusion that is detected and signaled by the applet. This combined attack is an adaptation of the attack presented in Section 2.1 to the specificity of the embedded Java Card virtual machine highlighted in section 2.2.

The preparation of the attack is almost similar. Firstly, the attack applet declares the two classes A and B presented in Figure 5. Secondly, the permanent memory is filled with many instances of class *B* and a single instance of class *A*. All the fields of all the *B* instances are initialized to point to the unique *A* instance. Figure 6 represents the persistent memory state and the instance pool after object initialization.

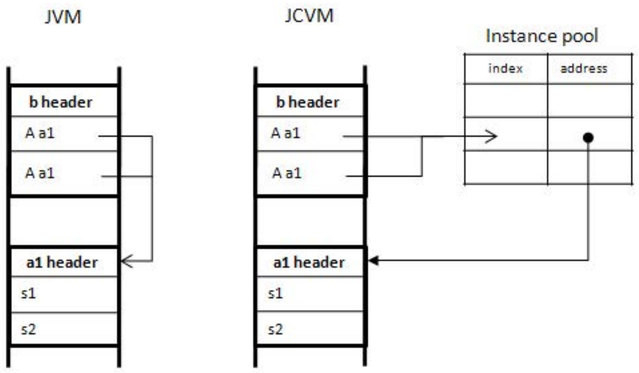

Fig. 4. Java and Java Card virtual machines addressing mode

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

Fig. 5. Classes of our applet

The algorithm of the applet performs the following operations. The applet loops through all *A* fields of all the objects in memory and checks whether they still contain a reference to the unique A instance. When a perturbation (i.e. light fault injection) flips a bit in persistent memory, it will likely change the internal reference of a A field as the main part of the memory is filled with A field. These internal references are indexes in the instance pool of the Java Card virtual machine. Because all instances created by the applet are of type B except the only A instance, the dereference of this erroneous index will return an instance of class B provided it is inside the instance pool boundaries.

In opposite to the original attack, the type confusion is performed in a single step (see Listing 1.2). When the attack success is detected through the Java pointer equality test (let's say on the b31.a2 field), the resulting dereferencing produces a type confusion because the reference aObj is statically typed as an object of type A, whereas it references an object of type B.

Listing 1.2. Single step type confusion on an erroneous field reference

```
A aObj; B b31;
aObj = b31.a2;      // type confusion
```

It should be noted that the attack succeed when the fault injection flips any bit of the instances allocated by our applet, except the B instances headers and the only A instance. In consequence, the number of instances allocated by the applet and the ratio between the object headers and the object fields have a strong impact on the success rate of the attack. This aspect will be discussed further in Section 3.

When the Java pointer equality test between an A field and the unique A instance fails, the type confusion is successful. The applet exits the main loop and outputs a specific status word to indicate the attack success. The type confusion can then be exploited through the erroneous A field.

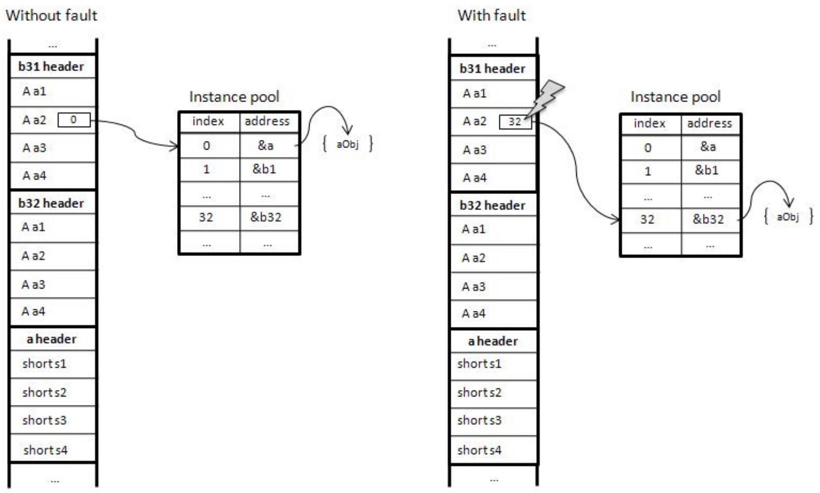

Fig. 6. Realization of type confusion on a Java Card virtual machine

Exploitation of the Type Confusion. The combined attack presented so far produces a reference of type A (aObj) that references an instance of type B (b32 in our previous example). The exploitation of the type confusion consists in accessing the fields of the B instance through the aObj reference and through the b32 reference.

However, when the fault injection switches the A field index, the new index can reference any B instance on the card. It is therefore necessary to identify the B instance that is referenced by the aObj reference. The easier way to obtain this information would be to code the index of the B instance in a field of the B object, but this approach has a major drawback. The bytes used to code the index in B instances would not produce a type confusion if switched through fault injection, thus lowering the chances of success of our attack.

The exploitation of the type confusion can be realized without impacting the attack success rate by realizing a second, indirect type confusion. After the first type confusion has been successfully realized, the aObj variable references the same object as the b32 variable which means that aObj.s1 and b32.a1 fields reference the same memory slot. In order to identify the B object that is referenced by the aObj variable, we set the aObj.s1 field to an arbitrary value, which produces a new type confusion as the equivalent b32.a1 field doesn't point to the unique A instance anymore. The applet then loops through all *A* fields except the one referenced by aObj and finds through Java pointer equality the one that differs from the unique A instance. This *A* field belongs to the B instance we can use to exploit our type confusion (b32 in our previous example).

Once the both references of type A and of type B that reference the same instance have been identified, the type confusion can be exploited to provide illegal access to the card memory. The exploitation of the type confusion is synthesized in Figure 7. The aObj.s1 field is set to forge an index of the Virtual machine instance pool. This index is dereferenced as an instance of type A through the equivalent b32.a1 field. The short fields of this instance (b32.a1.s1 to b32.a1.s4) can then be accessed in reading and writing to dump and modify arbitrary memory slots.

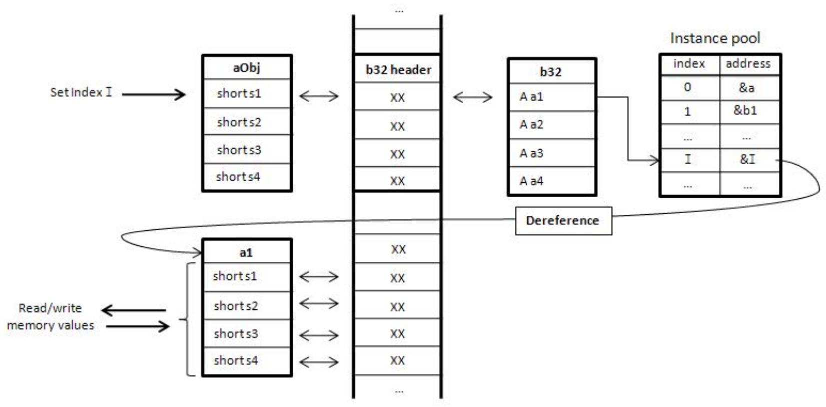

Fig. 7. Exploitation of type confusion on a Java Card virtual machine

3 The Practical Attack

3.1 Optimal Parameters

The goal of our attack is ultimately to raise the chances of success of the fault injection part of a combined attack by evading the localization constraints of the physical attack. This is made possible through a specific allocation of objects that creates a favorable pattern in persistent memory. Therefore, as we've seen in the section 2.3, the design of the classes have a strong impact on the success rate of the attack. The optimal ratio is not trivial as two orthogonal factors must be taken into account :

- The number of instances, that produce overhead through their headers,
- The number of fields, that produce overhead through the applet code necessary to test Java pointer equality.

In order to exploit the full possibilities of this attack, we determine the optimal ratio between these two factors. In the remainder of this paper, we use the following notation :

- A (constant) : allocated persistent memory space before applet loading, in bytes,
- B (constant) : size of the bytecode necessary to test a single reference through Java pointer equality, in bytes,
- C (constant) : size of the applet, except the bytecode necessary to test the references, in bytes,
- D (constant) : total size of the persistent memory, in bytes,
- x : number of instances
- y : number of references per instance
- xy : total number of references in persistent memory

Using these notations, we define the probability of hitting a reference with a random fault injection as the ratio between the size of references in memory and the total size of memory (provided references are coded on two bytes):

$$p(hit) = \frac{2xy}{D} \tag{1}$$

In addition, provided instance headers are coded on two bytes, we can decompose the persistent memory as follows:

$$D = 2xy + A + C + By + 2x \tag{2}$$

As a result, we can express the number of instances in function of the number of references per instance:

$$y = \frac{D - A - C - 2x}{2x + B} \tag{3}$$

This gives us the probability of hitting a reference with a fault injection in function of the number of instances:

$$p(hit) = \frac{2x(D - A - C - 2x)}{(2x + B)D} \tag{4}$$

We apply our approach on a chip with 80 kBytes of flash persistent memory. The size of our applet is 5005 bytes except the reference test bytecode, that represents 17 bytes per reference. The allocated persistent memory before we load the applet is 9567 bytes. We compute the optimal ratio between the number of instances and the he number of fields using the following parameters :

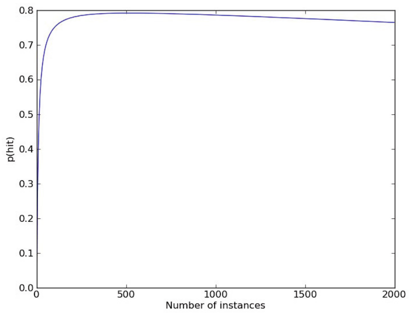

Fig. 8. Probability of hitting a reference with a fault injection in function of the number of instances

- A = 9567 bytes
- B = 17 bytes
- C = 5005 bytes
- D = 80 kBytes

The resulting probability of hit is shown on figure 8. This function is maximal for 519 instances. Using the formula 3, we determine that each instance should have 61 reference fields. As a result, we obtain a probability of hitting a reference with a random fault injection of 79%.

$$p(hit) = 0.79$$

3.2 Attack Simulation

In order to validate our approach, we have built a fault simulator as a plugin on the reference implementation provided by Oracle. Our plugin generates errors in the reference implementation's representation of the persistent memory and monitors the resulting behaviour of the applet. After the behaviour caused by the fault injection has been analyzed, the persistent memory is restored and another fault is generated. We consider only byte-fault models for the fault injection simulation, and we generate faults according to two different models :

- Byte modification fault model : the byte affected by the fault is replaced by an arbitrary hexadecimal value,
- Stuck-at fault model : the byte affected by the fault is replaced by either 0x00 or 0xFF hexadecimal value.

In addition, the fault simulator supports two fault localization model:

- Random localization model : the location of the fault is chosen randomly ,
- Scanning localization model : each byte of the memory is faulted one after the other.

We perform our attack simulation using the byte modification fault model. This fault model produces a superset of the faults generated by a bit-flip fault model which is representative of the effects of a fault injection on the persistent memory during a read operation. The Figure 9 shows maps of the smartcard's non-volatile memory where successful injections are represented as grey dots. The Figure on the left presents the simulation of an 8000 laser pulses campaign using a random localization model and a byte modification fault model, while the Figure on the right presents the simulation of a 64000 laser pulses campaign using a scanning localization model and a byte modification fault model.

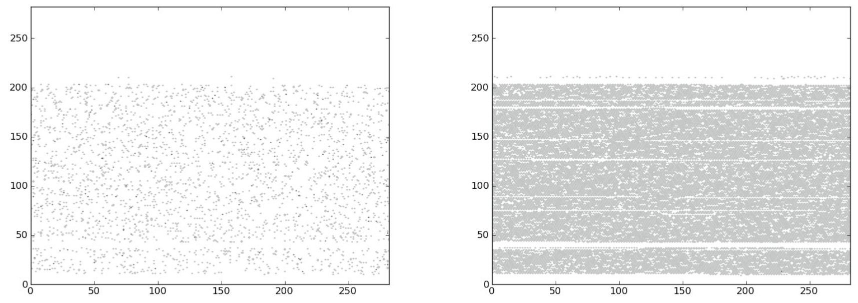

Fig. 9. EEPROM map of successful injections using a random localization model for an 8000 laser pulses campaign (left) and a scanning localization model for a 64000 laser pulses campaign (right)

The results obtained by simulation show that a large part of the non volatile memory produces a type confusion when exposed to external perturbation. The part of the non-volatile memory that does not produces type confusions is mainly filled with the applet's code (top white part of the graphic) and the virtual machine's persistent objects headers (white linear patterns). We can conclude that the presented attack fully aims its objective as most of the fault injections produce a type confusion, regardless of the localization of the laser pulse in the non-volatile memory.

4 Countermeasures

4.1 Defensive Virtual Machines

The Java Card specifications ensure that well-typed code executes identically on every implementation of the virtual machine. However, when it comes to ill-typed code, the specifications leave a lot of freedom concerning the implementation of dynamic checkings. Because ill-typed code is out of the scope of the Java Card specifications, some virtual machines are much more defensive than others, and the behavior in presence of ill-typed code can vary a lot. An overview of possible dynamic runtime checks is provided in [18]. Among the dynamic runtime checks implemented in Java Card virtual machines, the firewall checks can provide a defense against type confusion exploitation. These checks are enforced during object dereferencing and prevent access to objects belonging to other contexts.

However, our experiments with ill-typed code on modern virtual machines have shown that our attack can nevertheless grant illegal memory access. Indeed, firewall context checks are usually performed on metadata stored in the object's header. When our fault injection succeeds, the header of an A instance is assigned to an arbitrary memory slot whose data can be interpreted as the header of an instance belonging to our applet or even to the JCRE. In this case, the memory access is granted by the virtual machine.

When the defensive Java Card virtual machine enforces stronger defenses, the type confusion can be exploited in another way. The class A and the class B can be appended an additional field whose only purpose is to exploit the type confusion once the fault injection has succeeded. For example, the modified classes presented in Figure 10 allow the attacker to access the *tabConfusion* byte array as a short array (which have been proven to be extremely harmful, as presented in [14]). The type of this additional field can be chosen by the attacker to defeat the defences implemented in the virtual machine. However, this solution has a major drawback: the bytes used to code this additional field would not produce a type confusion if switched through fault injection. These bytes are only useful for the exploitation of the type confusion, and thus lower the chances of success of the fault injection.

```
class A {                        class B {
short s1;                          A a1;
short s2;                          A a2;
short s3;                          A a3;
short s4;                          A a4;
short[] tabConfusion;              byte[] tabConfusion;
};                               };
```

Fig. 10. Classes of our applet modified to cope with defensive virtual machines

More generally, combined attacks are a mean of embedding malicious code inside a legal applet whose attack load is activated by fault injection. Therefore, our combined attack can not succeed on defensive virtual machines that enforce efficient dynamic countermeasures against ill-typed code execution.

4.2 Memory Protection

Besides virtual machines countermeasures, some secured IC embed hardware and software memory protections that allow detection of memory errors.

These protections include redondant read operations in persistent memory to detect inconsistencies, parity checking and error-correcting codes to detect and correct errors in persitent memory.

When activated, such countermeasures are efficient at detecting and preventing the fault injections described in this paper.

5 Related Works and Conclusion

5.1 Related Works

Our work is an adaptation of the attack presented in [7] to the domain of smart cards. The specificity of the Java Card applets embedded on smart cards have already been presented in section 2.2. This specificity prevents exploitation of the attack originally designed for classical virtual machines on state-of-the-art Java Card virtual machines. In addition to adapt the original attack to the smart card domain, we also prove the efficiency of our approach through fault simulation.

Other publications [29,4] present combined attacks that exploit both malicious applets and fault injection to activate the attack load of the applet on-card, thus luring the bytecode verifier. However, unlike the attack we present in this paper, these attacks demand a high precision in the localization of the fault injection. Inversely, our attack allows to evade the localization constraint of the fault injection when performing the combined attack.

Finally, in [3], Barbu et. al. propose a combined attack that rely on the multi-threading capacities of the Java Card 3.0 connected edition platform to interrupt the virtual machine execution and thus evade the timing constraint in the fault injection. Like our attack, their work eases the fault injection part of the combined attack by lowering the precision requirements of the laser pulse. However their attack targets a platform that is barely deployed nowadays, while ours targets the main stream Java Card 2 as well as the Java Card 3 standard edition platforms.

5.2 Conclusion

The emergence of Java Card 3 standard edition platforms with embedded bytecode verifier leads the security community to design new types of attacks that combine both software and physical attacks. These attacks, called combined attacks, inherit the constraints of the classical fault injection attacks: they require a high degree of precision in the timing and the localization of the physical fault injection.

As the smartcard market keeps growing, the security of the integrated circuits is getting more and more efficient. Upscale chips now embed a high range of invasion sensors that prevent attackers to compromise the security of the applications. Therefore, scanning the whole circuit is not an option anymore.

The combined attack paradigm presented in this paper allows to lower drastically the requirements on the localization of the physical fault injection. As a result, the attacker can choose any physical zone that shows less security, without trading the success rate of the combined attack. The resulting type confusion have been proved to be extremely compromising for a large range of nowadays Java Card platforms.

References

1. Agrawal, D., Archambeault, B., Rao, J., Rohatgi, P.: The EM side-channel(s). In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 29–45. Springer, Heidelberg (2003)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks (2004)
3. 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)
4. 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)
5. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
6. Giraud, C., Thiebeauld, H.: A survey on fault attacks. In: Quisquater, J.-J., Paradinas, P., Deswarte, Y., El Kalam, A.A. (eds.) Smart Card Research and Advanced Applications VI. IFIP, vol. 153, pp. 159–176. Springer, Boston (2004)
7. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: IEEE Symposium on Security and Privacy, pp. 154–165. IEEE Computer Society (2003)
8. Hogenboom, J., Mostowski, W.: Full memory read attack on a Java Card (2003)
9. Iguchi-Cartigny, J., Lanet, J.L.: Developing a trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
10. Kim, C.H., Quisquater, J.J.: Faults, injection methods, and fault attacks. IEEE Des. Test 24, 544–545 (2007)
11. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
12. Kömmerling, O., Kuhn, M.G.: Design principles for tamper-resistant smartcard processors. In: Proceedings of the USENIX Workshop on Smartcard Technology, p. 2. USENIX Association, Berkeley (1999)
13. Lancia, J.: Un framework de fuzzing pour cartes à puce: application aux protocoles EMV. In: Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC, pp. 350–368 (2011)
14. Lancia, J.: Compromission d'une application bancaire JavaCard par attaque logicielle. In: Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC (2012)
15. Lindholm, T., Yellin, F.: The Java(TM) Virtual Machine Specification, 2nd edn. Prentice Hall PTR (April 1999)
16. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Examining smart-card security under the threat of power analysis attacks. IEEE Trans. Comput. 51, 541–552 (2002)
17. Moore, S., Anderson, R., Cunningham, P., Mullins, R., Taylor, G.: Improving smart card security using self-timed circuits. In: Technology, Fourth AciD-WG Workshop, Grenoble, ISBN, pp. 211–218 (2002)

18. 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)
19. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In: Attali, I., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 200–210. Springer, Heidelberg (2001)
20. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
21. Skorobogatov, S.P.: Semi-invasive attacks – a new approach to hardware security analysis. Tech. Rep. 630, University of Cambridge, Computer Laboratory (April 2005), `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf`
22. Sun Microsystems, Inc., Palo Alto/CA, USA: Java Card Platform Security, technical White Paper (2001)
23. Sun Microsystems, Inc., Palo Alto/CA, USA: Java Card 2.2 Application Programming Interface, API (2002)
24. Sun Microsystems, Inc., Palo Alto/CA, USA: Java Card 2.2 Runtime Environment (JCRE) Specification (2002)
25. Sun Microsystems, Inc., Palo Alto/CA, USA: Java Card 2.2 Virtual Machine Specification (2002)
26. Sun Microsystems, Inc., Palo Alto/CA, USA: Application Programming Interface (API) - Java Card(TM) Platform, Version 3.0.1 (2009)
27. Sun Microsystems, Inc., Palo Alto/CA, USA: Runtime Environment Specification - Java Card(TM) Platform, Version 3.0.1 (2009)
28. Sun Microsystems, Inc., Palo Alto/CA, USA: Virtual Machine Specification - Java Card(TM) Platform, Version 3.0.1 (2009)
29. 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)