Heap . . . Hop! Heap Is Also Vulnerable

Guillaume Bouffard, Michael Lackner, Jean-Louis Lanet, Johannes Loinig

Heap . . . Hop!
Heap Is Also Vulnerable

Guillaume Bouffard[1,2(✉)], Michael Lackner[3], Jean-Louis Lanet[4], and Johannes Loinig[5]

[1] University of Limoges, 123 Avenue Albert Thomas, 87060 Limoges, France
guillaume.bouffard@ssi.gouv.fr
[2] Agence Nationale de la Sécurité des Systèmes D'Informations, 51, Boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France
[3] Institute for Technical Informatics, Graz University of Technology, Graz, Austria
michael.lackner@tugraz.at
[4] INRIA LHS-PEC, 263 Avenue Général Leclerc, 35042 Rennes, France
jean-louis.lanet@inria.fr
[5] NXP Semiconductors Austria GmbH, Gratkorn, Austria
johannes.loinig@nxp.com

Abstract. Several logical attacks against Java based smart card have been published recently. Most of them are based on the hypothesis that the type verification was not performed, thus allowing to obtain dynamically a type confusion. To mitigate such attacks, typed stack have been introduced on recent smart card. We propose here a new attack path for performing a type confusion even in presence of a typed stack. Then we propose using a Fault Tree Analysis a way to design efficiently counter measure in a top down approach. These counter measures are then evaluated on a Java Card virtual machine

Keywords: Java Card · Logical attack · Transient persistent heap · Counter measures

1 Introduction

Today most of the smart cards are based on a Java Card Virtual Machine(JCVM). Java Card is a type of smart card that implements the standard Java Card 3.0 [18] in one of the two editions "Classic Edition" or "Connected Edition". Such a smart card embeds a virtual machine, which interprets application byte codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [9]. This protocol ensures that, the code owner has the required credentials to perform the particular action.

A smart card can be viewed as a smart and secure container which stores sensitive assets. Such tokens are often the target of attacks at different levels: pure software attacks, hardware based, *i.e.* side channel of fault attacks but

M. Joye and A. Moradi (Eds.): CARDIS 2014, LNCS 8968, pp. 18–31, 2015.
DOI: 10.1007/978-3-319-16763-3_2

also mixed attacks. Security issues and risks of these attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. This requires a clear understanding and analysis of possible attack paths and methods to mitigate them through adequate software/hardware countermeasures. The current smart cards are now well protected against pure logical attacks with program counter bound checks, typed stack and so on. For such smart cards, we propose in this paper, two new attacks that target the heap of the JCVM. The first one is on the transient heap while the second allows a type confusion on the permanent heap.

Often countermeasures are designed in a bottom-up approach, in such a way that they cut efficiently the attack path but a new avatar of an attack path can be found easily. We propose here to use a top down approach to mitigate the attack by protecting the assets instead of blocking the attack path.

The remaining of the paper is organized as follows: the second section introduces the related works on logical attacks. The third section presents our contributions on the heap: the transient array and the type confusion. Then, in the fourth section, we propose some counter measures designed with a top down approach and we evaluate them in term of performance. Finally, in the last section, we conclude.

2 State of the Art of the Logical Attacks

Logical attacks are based on the fact that the runtime relies on the Byte Code Verifier (BCV) to avoid costly tests. Then, once someone find an absence of a test during runtime, there is a possibility that it leads to an attack path. An attack aims to confuse the applet's control flow upon a corruption of the Java Card Program Counter or perturbation of the data.

2.1 Fooling the Control Flow Graph

Misleading the application's control flow purposes to execute a shellcode stored somewhere in the memory. The aim of EMAN1 attack [12], explained by Iguchi-Cartigny et al., is to abuse the Firewall mechanism with the unchecked static instructions (as `getstatic`, `putstatic` and `invokestatic`) to call malicious byte codes, this behavior is allowed by the Java Card specification. In a malicious CAP file, the parameter of an `invokestatic` instruction may redirect the Control Flow Graph (CFG) of another installed applet in the targeted smart card. The EMAN2 [6] attack was related to the return address stored in the Java Card stack. They used the unchecked local variables to modify the return address, while Faugeron in [8] uses an underflow on the stack to get access to the return address.

When a BCV is embedded, installed an ill-formed applet is impossible. To bypass an embedded BCV, new attacks exploit the idea to combine software and physical attacks. Barbu et al. presented and performed several combined attacks such as the attack [3] based on the Java Card 3.0 specification leading to

the circumvention of the Firewall application. Another attack [2] consisting of tampering the Application Protocol Data Unit (APDU) that leads to access the APDU buffer array at any time. They also discussed in [1] about a way to disturb the operand stack with a combined attack. It also gives the ability to alter any method regardless of its java context or to execute any byte code sequence, even if it is ill-formed. This attack bypasses the on-card BCV [4]. In [6], Bouffard et al. described how to change the execution flow of an application after loading it into a Java Card. Recently, Razafindralambo et al. [20] introduced a combined attack based on fault enabled viruses. Such a virus is activated by hitting with a laser beam, at a precise location in the memory, where the instruction of a program (virus) is stored. Then, the targeted instruction mutates one instruction with one operand to an instruction with no operand. Then, the operand is executed by the JCVM as an instruction. They demonstrated the ability to design a code in a such way that a given instruction can change the semantics of the program. And then a well-typed application is loaded into the card but an ill-typed one is executed. Hamdouche et al. [11] introduced a mutation analysis tool to check the ability of an application to come a malicious one.

Hamadouche et al. [10] described various techniques used for designing efficient viruses for smart cards. The first one is to exploit the linking process by forcing it to link a token with an unauthorized instruction. The second step is to characterize the whole Java card API by designing a set of CAP files which are used to extract the addresses of the API regardless of the platform. The authors were able to develop CAP files that embed a shellcode (virus). As they know all the addresses of each method of the Application Programming Interface (API), they could replace instructions of any method. In [20], they abuse the on board linker in such a way that the application is only made of tokens to be resolved by the linker. Knowing the mapping between addresses to tokens thanks to the previous attack, they have been able to use the linker to generate itself the shellcode to be executed.

We have presented attacks which perturb the application's control flow. Cheating the CFG leads to execute malicious bytecode or prevent any instruction to correctly finish. Another approach is exploiting the Java Card heap to access to unauthorized fields.

2.2 Exploiting the Java Card Heap

Lancia [13] exploited the Java Card instance allocator of Java Card Runtime Environment (JCRE) based on high precision fault injection. Each instance created by the JCRE is allocated in a persistent memory. The Java Card specification [18] provides some functions to create transient objects. The data of the transient object are stored in the RAM memory, but the header of this object is always stored in the persistent memory. On the modern Java Card using Memory Management Unit (MMU), references are represented by an indirect memory address. This address is an index to a memory address pool which in turn refers to a global instance pool managed by the virtual machine.

In this section, we have introduced logical attacks on Java Card from the literature. In this paper, we focus on the heap security. In the next section, I will present new ways to break the Java Card heap integrity.

3 Logical Attacks Against the Java Card Heap

From the state of the art, reading the memory needs to write at least 2 bytes to read few bytes. This method stresses the memory and will need more than 65,000 writing to the same cell. So 10 or 20 executions of a shellcode will kill the card reaching the stress threshold of the EEPROM. We need to have a smarter shellcode. To improve this approach we purpose to use transient array.

A transient array is an array where the data are stored in RAM and its descriptor is stored in EEPROM, precisely in the owner's heap area. Thus a transient array lost its content during power off but not the reference, there is no *natural* garbage collection. Unlike the EEPROM, one can write indefinitely in RAM area. So, using a transient array is better to dump RAM and EEPROM parts to avoid memory stress. To understand how a transient array is stored in the smart card, we created a simple applet which gets the transient array address and reads data at this address.

3.1 Transient Arrays on Java Card

So, using a transient array is better to dump RAM and EEPROM parts to avoid memory stress. To understand how a transient array is stored in the smart card, we created a simple applet which gets the transient array address and reads data at this address.

An implementation of the transient array's header is the following at the EEPROM area address:

```
0x8E85: 0x00 0x04 0x5B 0x30 0x6C 0x88 0x00 0x0A 0x05 0xB9
```

Where `0x0004` is the size of the structure without the metadata corresponding to the header. In the header part the byte `0x5B` corresponds to the transient byte array type. The three next bytes are probably the security context `0x30 0x6C 0x88`. It remains the four last bytes as pseudo data. After several experimentation, we understood that `0x000A` represents the size of the data in RAM and `0x05B9` its address as shown in the Fig. 1.

We have disclosed how a transient array is design in a card implementation, focus on how to modify one. Confusing one purposes us to read and write anywhere in the memory.

3.2 Type Confusion Upon the Java Card Heap

The Java Card heap contains the runtime data for all class instances and allocated arrays. The instance data can be a reference to an object, an instance of

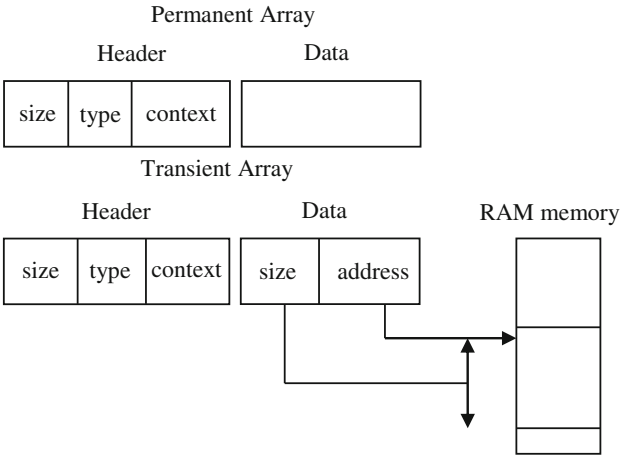

Fig. 1. Structure of transient and permanent arrays.

a class or a numerical value. In the Java Card architecture, the heap is a persistent element stores in the EEPROM area. Due to the limited resources, the instance data are often not typed. To have access to the instance fields, the Java Card specification [18] defines `getfield_<t>_this` and `putfield_<t>_this` as typed instructions on a `t` typed element. The type `t` can be a reference (`<t>` is replaced by `a`), a short value (type is `s`), etc. The `getfield_<t>_this` instruction pushes the field value onto the stack. On the opposite, the `putfield_<t>_this` instruction stores the latest pushed value. From the stack point of view, the last element must be a `t` type.

Latest smart cards based on Java Card technology increasingly implement typed stack. To succeed a type confusion on this kind of platform, I propose to exploit the untyped instance fields. Let us assume the code shown in the Listing 1.1. On a card which no embeds any BCV, this method aims at converting a given reference given in parameter to a short value returned.

Listing 1.1. Receive reference of an object by type confusion over instance fields.

```
short getObjectAddress (object object) {
  02 // flags: 0 max_stack : 2
  12 // nargs: 1 max_locals: 2
  /*005f*/ L0: aload_1 // object reference given in
     parameter
  /*0060*/      putfield_a_this 0
  /*0062*/      getfield_s_this 0
  /*0064*/      sreturn
}
```

In the Listing 1.1, the field `0` is accessed as a reference (at `0x60`) and as a short value (at `0x62`). In the case of the use of a typed stack, only two types are

supported, the short and reference types. The `putfield_a_this` instruction (at `0x60`) saved the value given in parameter into the field `0`. The `getfield_s_this` (from `0x62`) pushes the value of the field `0` to stack as a short. A type confusion can then be performed on the instance fields. There, the reference given as parameter is then returned as a short value. From the Java Card stack side, the type of each manipulated element is correct. Nonetheless, a type confusion has been performed during the field manipulation.

In this section, we have explained a new typed confusion attack on Java Card smart cards which embed typed stack. As the stack mechanism cannot be confused, we focused on the instance fields which are often untyped. Thus, the type confusion attack moves on the Java Card stack to the instance fields.

3.3 Setting up Transient Array Metadata to Snapshot the Memory

Based on the function shown in the Listing 1.1, we are able to update this reference to point out a fake transient array. Stored in a Java array, we succeed in retrieve its reference upon on the type confusion explained in the previous section. On the targeted platform, each transient array has the same metadata's pattern. The transient array's header can be so update with our properties.

Some cards prevent accessing to transient array out of a specific heap area. On the targeted card, a typed stack is embedded but no BCV. So, using static instruction abuse of the firewall [12] to read and write anywhere in memory, as shown in the Listing 1.2. This code can be executed through the EMAN2 [6] attack. Assume that the transient array size and the data address are located from `0x8E9D`.

Listing 1.2. Executing the basic shellcode

```
18 FF     sspush 0x00FF
80 8E 9B  putstatic_s 0x8E9B //size: 0x00FF
18 00     sspush 0x00FE
80 8E 9D  putstatic_b 0x8E9D //address: start from 0x00FE
7A        return
```

There, we are able to set the size and the address of our transient data to cover from `0` for `0xFFFF` bytes, *i.e.*, the whole memory. This behavior is accepted by the targeted card and this corrupted transient array can be used to read the complete memory. Fooling transient array is more efficient than the attacks presented in the state of the art: we need to write only few bytes in memory to obtain an array which can be read normally.

Once this shellcode is executed, we have to copy the array in the APDU buffer slicing it into slots of 255 bytes to fit the size of the APDU buffer. Unfortunately, the ROM is always unread by this approach. The values returned at the ROM area are filled with 0. With attack as EMAN2 [6], the dumping shellcode needs to write around 65,000 times into a particular cell. There, we have improved the dump with only one write into each cell for 255 read bytes. We reduced greatly the execution time[1] and minimized the memory stress.

[1] Writing in EEPROM needs to erase which is time consuming.

4 Countermeasures

The security of the Java Card sandbox model is threaten by two main types of attacks. The first are, as used by the proposed attack of this paper, logical attacks by uploading malicious applets. The second class are fault attacks (laser beam) which threaten the integrity of the memory.

4.1 Counteract Fault Attack on the Java Heap

The common fault attack model, which is also used in this work, is that an adversary can set[2] bytes inside the card memory to `0x00` or `0xFF`. This model is called *precise byte error* and is presented in Table 1. The difficulty for an attacker to set bits inside the card to either `0x0` or `0x1` is called *precise bit error* and is currently no realistic fault model.

Table 1. Current fault models to evaluate possible countermeasures and security threats on Java Cards [7, with modifications].

Fault Model	Precision	Location	Timing	Fault Type	Difficulty
precise bit error	bit	precise control	precise control	BSR[a], random	++
precise byte error	byte	loose control	precise control	BSR, random	+
unknown byte error	byte	loose control	loose control	BSR, random	-
random error	variable	no control	loose control	random	–

[a] bit set or reset.

The transient array objects in the heap contain the size (2 bytes) and start address (2 bytes) of the array fields. Due to the *precise byte error* fault model an adversary is able to set the size field to `0xFFFF`. This enables again a full memory dump of the RAM even if no malicious applet is installed. Therefore, it is a substantial need to protect the array object headers against fault attacks.

Fault attacks can either inject transient faults or permanent faults into the memory. An industrial often used countermeasure against these transient faults are multiple readings from the same address and the comparison if all read-out values are equal. The change for a successful attack by circumventing the multiple readings by additional fault attacks is a negative exponential distribution.

Unfortunately, a multiple read is no protection against an attacker which uses a strong enough laser to permanently change the values of a memory cell. A multiple read on a permanently changed memory cell always results in the same read-out value. Therefore, to counteract such a permanent fault, a statically calculated checksum is needed. This checksum is re-calculated during run-time and compared to the statically calculated one. Generally the checksum countermeasure, compared to multiple reads, consumes more run-time performance and requires additional non-volatile memory.

[2] Memory encryption results in a logical read-out value which is random.

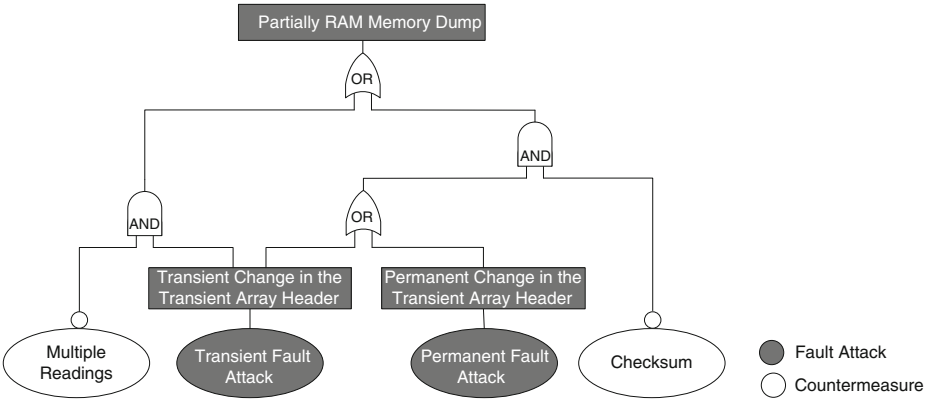

Fig. 2. Fault tree overview of the different possible attack paths to fulfill a partial memory dump of the RAM by a fault attack. Furthermore, effective countermeasures against the two general types of fault attacks are listed.

In summary to counteract transient fault attacks we propose to perform multiple readings on the accessed object header elements. To counteract permanent and transient faults on the object header we propose as countermeasure a checksum. An overview of the attack paths and countermeasures against these fault attacks is shown in Fig. 2. Multiple readings on the object header counteract transient fault attacks. Checksums counteract transient faults and permanent faults on the object header.

4.2 Logical Attack on the Java Heap

Unfortunately, the proposed checksums and multiple reads over the object headers in the Java heap, to counteract fault attacks, is not an effective countermeasure against logical attacks. By a logical attack it is quite easily possible to study the algorithm of the checksum creation and create valid checksums for manipulated object headers. Therefore, other countermeasures must be found to counteract logical attacks.

To find an appropriate countermeasure an attack tree for the proposed attack of this work is shown in Fig. 3. Starting from the lower left it is shown that the causes of the execution of illegal bytecodes can be either a logical attack with the absence of an on-card BCV or a fault attack. The presence of illegal bytecodes is a cause to successfully perform a type confusion attack between *integral data* or *reference*. The type confusion can be either performed on the operand stack or, as proposed in this work, on instance fields of objects. This type confusion is the first main requirement for the full RAM memory dump attack of this work. The second requirement is the manipulation of the integrity of the transient array object header. This integrity violation can be reached by various kind of the proposed attacks EMAN1 [12], EMAN2 [6], and EMAN4 [6].

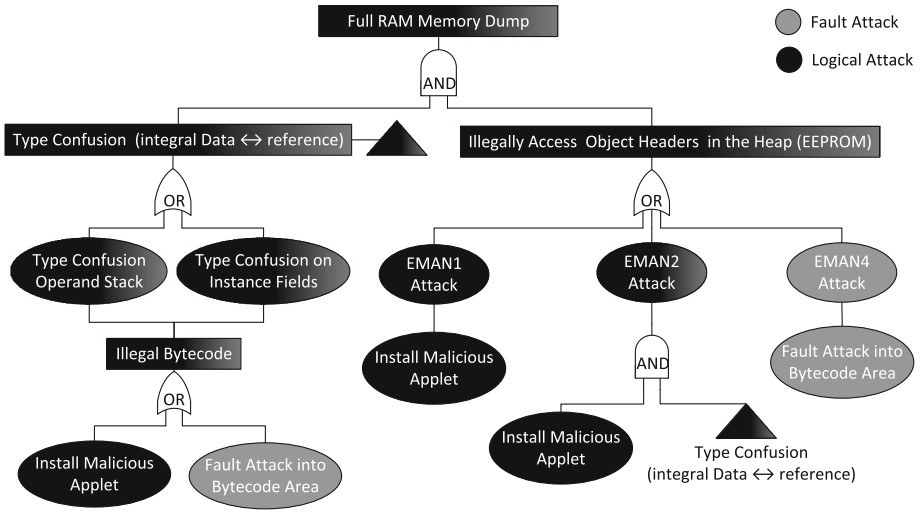

Fig. 3. Fault tree overview of the different possible attack paths and needed attack preconditions to fulfill the memory dump attack of this work.

EMAN1 relies on the fact that illegal static variables are not checked during run-time by the Java Card firewall. EMAN2 relies on the fact that it is possible to overwrite the Java frame return address by a bytecode with an illegally index of bytecode accessing the local variables. The return address is overwritten with the address of a Java array which is previously received by a type confusion attack. EMAN4 relies on a fault attack during run-time which illegally changes the operands of a `goto_w` bytecode. This attack results in a jump and execution of a Java array filled with malicious bytecodes.

To install applets on Java cards a secret key must be known which is only available for authorized authorities. Nohl [16] presented in 2013 an attack to crack this key (DES encryption) which enables the installation of malicious applets. Based on Nohls presented information the industry created guidelines to counteract this attack. Therefore, we assume that the installation of malicious applets on currently available credit cards or bank cards is again a very unlikely security threat.

Therefore, half of the starting points of the attacks of this work, previously presented in Fig. 3, are not available on industrially used Java Cards. Furthermore, Java Cards are becoming more and more powerful which will most probably result in an available on-card verification process which only accepts applets which only contain harmless operations. Resource optimized on-card verification algorithms are presented in different works [5,14].

Nevertheless, the attack preconditions of this work (type confusion and access to the object header) can be also reached by uploading a valid applet and turning them into a malicious one. This transformation is done by performing fault attacks into the bytecode area. Therefore, additional security mechanisms must

be integrated when operands or opcodes are fetched from the bytecode area. Various countermeasures [15,21–23] are proposed to protect the integrity of the bytecode area.

Protect Integrity of the Bytecode Area: The replacement of the typically not used bytecode `NOP` (`0x00`) is proposed in [23] to counteract the threat of skipping bytecodes. The authors of [15] create a fingerprint of an applet which is based on the position of critical bytecodes/values (`0x00`, `0xFF`, branch, jump, subroutine call, return) inside a method. This fingerprint is than checked during run-time. Another countermeasure against the illegal execution of arbitrary bytecodes is the encryption of the bytecode based on a secret key and the memory address where the bytecode is stored [21]. The authors in [22] propose to divide the bytecodes of a method into basic blocks. During an off-card preprocessing step checksums are calculated over these basic blocks and stored into the applet as an additional component. During run-time the checksums are re-calculated and compared to the off-card calculated checksums.

Based on the required level of security all of these countermeasures are a possible solution to counteract integrity attacks into the bytecode area. These attacks are needed as a starting point of an attacker to reach the goal of turning valid applets into malicious one. These malicious applets are the starting point to create the proposed memory dump attack of this work on industrially used Java Cards.

5 Experimental Results

The measurements of this work are based on a Java Card prototype implementation which is based on the Java Card specification [17,18]. We integrated our countermeasures into this prototype to counteract fault attacks which manipulate the array headers in the Java heap.

The JCVM is compiled with μVision3 which is a development tool especially for low cost processors based on the 8-bit 8051 platform. The performance results are based on the supplied memory-accurate 8051 instruction set simulator of the μVision IDE. The tested Java Card applets HelloWorld and Wallet are obtained from the official Java Card software development kit (SDK). The Calculator applet is self programmed. Note that the performance overhead measurements are normalized to a JCVM implementation which do not perform the additionally proposed countermeasures during run-time.

5.1 Fault Attack Countermeasures on the Object Header

Checksum: A checksum is statically calculated over the size and pointer element of each Java array header in the heap. The checksum is based on a XOR operation and has a length of one byte. Each array object header of our prototype, even the permanent arrays, contain a size and address field shown in

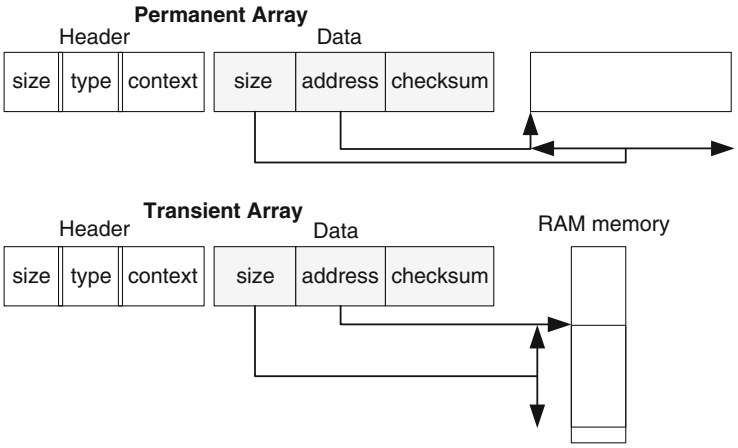

Fig. 4. Structure of transient and permanent arrays in the prototype implementation.

Fig. 4. For permanent arrays the address field points into the non-volatile memory (EEPROM). Therefore, the object header must be secured for permanent and transient arrays.

The checksum calculation and writing is performed during the execution of the `<t>newarray` bytecodes and the creation of a transient array by calling the Java Card API method `JCSystem.makeTransientByteArray()`. During runtime this checksum is re-calculated at each security-critical access to the array object header (e.g., `aaload`, `sstore`, `arraylength`).

Double Read: The double read is done when the size or pointer elements of the array object header are accessed by security-critical bytecodes. During the creation of the array header the write operation of the size and pointer element is checked by an immediate reading and comparison of the written values.

Execution Time Overhead: The run-time overhead of the checksum and double reads is shown in Fig. 5. The execution time of the `newarray` bytecode is quite long even if no additional security checks are performed which results in a low percentage overhead increase. Compared to this the `saload` bytecode, which loads a short value from an array, has an additional overhead of +9 % for double reads and +22 % for the checksum re-calculation.

The creation of a new Java array is in generally performed one time during the installation process of an applet. The overall applet execution time for different applets and bytecodes are presented in Table 2. The overall applet time measurements does include the sending of APDU commands for the selection of the applet, sending of commands to communicate with the applet, and the reception of results. Overall the additional checks do not significantly increase the overall execution time of the measured applets. The highest overall measured

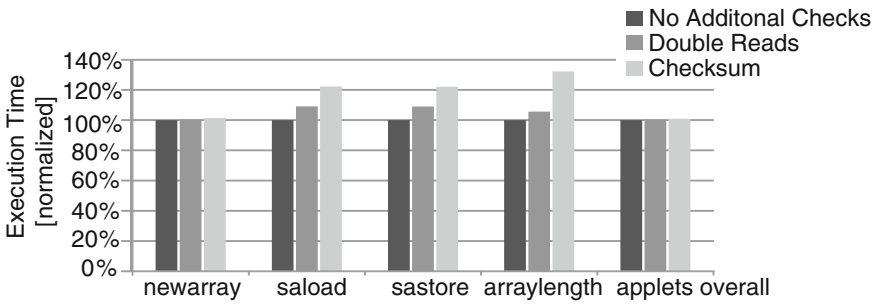

Fig. 5. Graphical representation of the performance impact of the additional double read and checksum calculations for selected bytecodes and the overall time of the measured applets.

time increase is only around +1 % for the self written Calculator applet and the checksum countermeasure. The double reads increase the Calculator applet by only around +0.5 %. Therefore, the higher security of the checksums, with regard to permanent memory faults, is paid with the price of one additional byte per array header and a doubling of the execution time overhead.

Table 2. Performance overhead overview of the double read and checksum countermeasures.

Java Bytecodes	Java Card Applet	Double Read	Checksum
`<t>newarray`		+1 %	+2 %
`<t>aload`		+9 %	+22 %
`<t>astore`		+9 %	+22 %
`arraylength`		+6 %	+22 %
	HelloWorld	+0.2 %	+0.5 %
	Wallet	+0.3 %	+0.9 %
	Calculator	+0.5 %	+1 %

6 Conclusion

Smart card designers now take into account the possibility to execute ill typed application even if the loaded applet is well typed. The combined attacks allow to use laser based attack to execute hostile applets. For this reason, designers protect dynamically the execution. Unfortunately, the attack paths can be subtle and the counter measures must protect the assets and not the attack paths. We presented two new attack paths that target the heap. The attack on the transient

array can be obtained via a laser on the size field. The exploitation allows to parse completely the memory without stressing the EEPROM. The second one exploits a type confusion even in presence of a typed stack.

We proposed, through the fault tree paradigm to perform a top down analysis to design the counter measures in order to improve their coverage. This approach avoid to mitigate different attack with several *ad-hoc* counter measures. We proposed different solutions implemented currently at the software level, but we plan to integrate them into hardware. We evaluated the cost in term of performances, the memory footprint being less important. The evaluation brought to the fore that such counter measures are affordable for the smart card domain.

References

1. Barbu, G., Duc, G., Hoogvorst, P.: Java card operand stack: fault attacks, combined attacks and countermeasures. In: Prouff [19], pp. 297–313
2. Barbu, G., Giraud, C., Guerin, V.: Embedded eavesdropping on java card. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 37–48. Springer, Heidelberg (2012)
3. 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)
4. 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)
5. Berlach, R., Lackner, M., Steger, C., Loinig, J., Haselsteiner, E.: Memory-efficient On-card Byte Code Verification for Java Cards. In: Proceedings of the First Workshop on Cryptography and Security in Computing Systems. CS2 2014, pp. 37–40. ACM, New York (2014)
6. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined software and hardware attacks on the java card control flow. In: Prouff [19], pp. 283–296
7. Dubreuil, J., Bouffard, G., Thampi, B.N., Lanet, J.L.: Mitigating Type Confusion on Java Card. IJSSE **4**(2), 19–39 (2013)
8. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 140–151. Springer, Heidelberg (2014)
9. GlobalPlatform: Card Specification. GlobalPlatform Inc., 2.2.1 edn., January 2011
10. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting byte code linker service to characterize java card api. In: Seventh Conference on Network and Information Systems Security (SAR-SSI), pp. 75–81 (22–25 May 2012)
11. Hamadouche, S., Lanet, J.L.: Virus in a smart card: Myth or reality? J. Inf. Secur. Appl. **18**(2–3), 130–137 (2013)
12. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. J. Comput. Virol. **6**(4), 343–351 (2010)
13. 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)
14. Leroy, X.: Bytecode verification on Java smart cards. Softw. Pract. Exper. **32**(4), 319–340 (2002)

15. Morana, G., Tramontana, E., Zito, D.: Detecting Attacks on Java Cards by Fingerprinting Applets. In: Reddy, S., Jmaiel, M. (eds.) WETICE, pp. 359–364. IEEE (2013)
16. Nohl, K.: Rooting SIM Cards. Speak at the Black Hat USA 2013 (2013)
17. Oracle: Java Card 3 Platform, Runtime Environment Specification, Classic Edition. No. Version 3.0.4, Oracle. Oracle America Inc., Redwood City, September 2011
18. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. No. Version 3.0.4, Oracle. Oracle America Inc., Redwood City (2011)
19. Prouff, E. (ed.): CARDIS 2011, vol. 7079. Springer, Heidelberg (2011)
20. Razafindralambo, T., Bouffard, G., Lanet, J.-L.: A friendly framework for hidding *fault enabled virus* for java based smartcard. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 122–128. Springer, Heidelberg (2012)
21. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.-L.: A dynamic syntax interpretation for java based smart card to mitigate logical attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Alcaraz Calero, J.M., Thomas, T. (eds.) SNDS 2012. CCIS, vol. 335, pp. 185–194. Springer, Heidelberg (2012)
22. Sere, A., Iguchi-Cartigny, J., Lanet, J.L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. Int. J. Secur. Appl. **5**(2), 49–61 (2011)
23. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.L.: Automatic detection of fault attack and countermeasures. In: Serpanos, D.N., Wolf, W. (eds.) WESS. ACM (2009)

Springer

http://www.springer.com/978-3-319-16762-6

Smart Card Research and Advanced Applications
13th International Conference, CARDIS 2014, Paris, France,
November 5-7, 2014. Revised Selected Papers
Joye, M.; Moradi, A. (Eds.)
2015, X, 261 p. 76 illus., Softcover
ISBN: 978-3-319-16762-6