A Friendly Framework for Hidding *fault enabled virus* for Java Based Smartcard

Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet

Secure Smart Devices (SSD) Team
XLIM/Université de Limoges – 123 Avenue Albert Thomas, 87060 Limoges, France
aina.razafindralambo@etu.unilim.fr,
{guillaume.bouffard,jean-louis.lanet}@xlim.fr

Abstract. Smart cards are the safer device to execute cryptographic algorithms. Applications are verified before being loaded into the card. Recently, the idea of combined attacks to bypass byte code verification has emerged. Indeed, correct and legitimate Java Card applications can be dynamically modified on-card using a laser beam to become mutant applications or *fault enabled viruses*. We propose a framework for manipulating binary applications to design viruses for smart cards. We present development, experimentation and an example of this kind of virus.

Keywords: Java Card, Virus, Logical Attack, Hidding Code.

1 Introduction

Nowadays, a new deployment model has been developed which has the ability to load third tier application in the SIM card through an application store controlled by the network operator. Unfortunately, these applications are being subjected to fault attacks as it is possible to design inoffensive applications, made hostile once hit by a laser beam. We call them *fault enabled viruses*. Our contribution is twofold, first we propose an architecture as tool and we provide a set of constraints to choose an instruction which will be subjected to a laser attack.

2 Context

Software attacks against smart card can be classified into two categories: ill-typed applications or well-typed applications. But the second category is again divided into permanent well-typed applications or transient well-typed applications. In ill-typed applications [9,4] the input file has been modified in order to illegally obtain information. Permanent well-typed application [8], relies on some weakness of the specification. Transient well-typed applications is a new research field [3,16,4] where an application mutes when a fault occurs. In this way, we have *fault enabled viruses*. Ill-typed applications and transient well-typed applications need to apply byte code transformation engineering at the CAP file level.

2.1   State of the Art

Physical Attacks. As explained by [2], a modification of the input current may modify the execution flow as the card is not self-powered as described in [1,10]. We also have attacks, explained by S. Skorobogatov and R. Anderson in [15], that use the light (LED, laser, *etc.*) and focus on a specific part of the chip, and the light provides enough energy in the memory-cell to change its value. Electromagnetic attack, presented in [13] and [14], as the inducted current provides a way to modify the memory value, and it also helps in characterizing the chip area used during a critical operation.

Logical Attacks. In E. Hubbers *et al.*'s paper [8], they presented a quick overview of the available classical attacks and gave some counter-measures. There are different way to get the type confusion: CAP file manipulation after the building step to bypass an off-card Byte Code Verifier (BCV); using fault injection to bypass the on-card one (difficult and expensive). There is also the use of the shareable interface mechanism, but on recent cards this attack is no longer possible. And finally, we have the transaction mechanism, that consists of making a set of atomic operations. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction and reset references to such objects as `null`. However, the authors found some cases where the card keeps the reference to the objects allocated during transaction even after a rollback. The idea of EMAN attack [9], explained by J. 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. In a malicious CAP file, the parameter of `invokestatic` instruction may redirect the control flow graph (CFG) of another installed applet in the targeted smart card. At CARDIS 2011, G. Bouffard *et al.* described, in [4], two methods to change the Java Card CFG. The EMAN2 attack will be further explained in the subsection 3.1.

2.2   The CAP File

As described by S. Hamadouche in [7], the CAP (`Convert APplet`) file format is based on the notion of interdependent components that contain specific information from the Java Card package. For instance, the `Method` component contains the methods byte code, and the `Class` component contains the information on classes such as references to their super-classes or declared methods.

3   The CapMap

3.1   Modification of a CAP File

CapMap has been developed [12] with the aim of having a handy and a friendly way to parse and modify a CAP file. It is very useful and very convenient while designing a logical attack to test Java Cards security. There are three steps to modify a CAP file using the CapMap: identifying which CAP file's components are located in our target, getting the right set of elements, and then applying

changes to the components; thanks to setters provided by the CapMap over each
CAP file elements. This is a simple example that makes the use of CapMap more
clear: it is a reference to the EMAN2 attack. We are going to use the CapMap
to particularly manipulate the instruction `sstore` to perform our attack. First,
we need to target our method within the `Method` Component, interdependent
to the other components. Element within it are indexed. A method is a set of
instructions, and an instruction is a set of byte-values. They both are indexed
in structures provided by CapMap. Secondly, to target the `sstore` instruction,
we are going to change its operand value. By changing the operand value we can
write in return function address as listing 1.1.

```
CapFileEditable  capFile = new CapFileEditable ();
capFile . load (MY_CAP_FILE);              // Load  the  cap  file
ArrayList<MethodInfo> methods =     // Get  methods
         capFile . getMethodComponent () . getMethods ();
//Set  the  instruction  you  want  to  replace
methods . get (METHOD_INDEX) . getBytecodes () . set
         (SSTORE_OPERAND_INDEX,  RETURN_ADDRESS_REGISTER) ;
```

Listing 1.1. CAP File modification with CapMap

3.2   Stack Evaluation

If the byte code of a java program is dedicated to be a *fault enabled virus* it
needs to avoid the software counter-measures embedded into the card. This type
of verification is performed for each method presented in the package. The type
checking ensures that no disallowed type conversion is performed. For example,
an integer cannot be converted into an object reference. A downcast can only be
performed using the `checkcast` instruction, and the arguments which are given
to the methods have to be compatible types. The most complicated step and
quite expensive (both time and memory), is to retrieve the type of local variables
by analyzing the byte code. It requires computing the type of each variable and
stack element for each instruction and each execution path, accepting programs
(set of instructions) where each stack element and local variable have the same
type whatever the path taken to reach an instruction. This also requires that the
stack size is the same for each instruction and for each path that can reach this
instruction. Another constraint is that the stack must never reach a maximum
size which allows checking, if we are not overflowing or underflowing the stack.
So, each time we modify a method we can verify the correctness type of the
modification. The most important thing for virus implementation is to define
the set of instructions eligible to be added to the byte array: only instructions
that are compatible with the previous instruction execution can be added to the
method. The type information associated to an instruction corresponds to the
type of the local variables and of the runtime stack **before the instruction
is executed**. The post conditions generated by the execution of the instruction
must be checked as pre-condition for the next instruction. This defines a set of
constraints that must be guaranteed by each byte code sequence.

3.3 Constraint Solving

To design a *fault enabled virus* we have to hide the real operation as a part of the operands of the preceding instruction. Thus, when the preceding instruction is hit by the laser and transformed as a `NOP` instruction: its operand becomes an instruction. Within this fault model, we need to find an instruction which needs one operand and satisfies several constraints, or an instruction which needs two operands. In such a case, the first operand becomes either the first instruction of the virus, or an instruction without operand and the second operand becomes the first instruction of the virus. We need to be able to select an instruction that satisfies several constraints, hence we will be able to hide viruses in a well-typed program. We try to build a sequence of instructions `prog`, empty at the beginning, such that it exists an instruction `ins`, with an operand number greater than one, for which the consumption of the stack is empty and the production on the stack is lower than the maximum value of the stack. If such an instruction exists, we can concatenate the sequence `prog` with the sequence `virus` minus its head. Executing the new sequence `prog` must lead to an empty stack at the end of execution. Unfortunately, the resulting program may be a non valid Java program: not all sequences of byte code can be generated by a compiler. But the certification scheme proposed by GlobalPlatform [5] do not indicate the source code. The certification process must be done at the CAP file level.

3.4 Java Card Code Reverser

The complete process of generating a *fault enabled virus* needs four steps using CapMap. Firstly, finding a sequence of instructions which hides the virus code that satisfies a set of constraints. The resulting CAP File represents a valid Java program in term of stack typing. Next, to evaluate the resulting cap file using an *off-card* BCV is the second step. If it is rejected, it means that either stack evaluation goes wrong, or the constraint solver failed. If the *off-card* BCV evaluation succeeds, the third step is, using our `Cap2Class` tool to reverse the code. Finally, converting the class file to `Java` file by means of existing tools, if the generated code is valid.

4 Evaluation of the Threat Capacities

4.1 Building a *fault enabled virus* with the CapMap

The listing 1.2 explains how to build the virus. It's aim is to send a clear text which has the value of an encrypted key container. Of course any analysis will reject this code as the secret key is being sent to the external world. This code can be split into three parts. The first one (`B1`) is mandatory and corresponds to the APDU reception. The second block (`B2`) corresponds to the code to obfuscate and which should only be executable once a fault occurs. It decrypts the key container and put the value in the APDU buffer at offset `0`. The last one (`B3`) sends the content of the apdu buffer from offset `0` for `16` elements (a 3-DES key) to the reader. If we can replace the `B2` block by an inoffensive code, it is said to be a *fault enabled smart card virus.* This code corresponds to the following byte code listed in 1.3.

```
public void process(APDU apdu) {
  short localS; byte localB;
  byte[] apduBuffer = apdu.getBuffer(); //get the APDU buffer
  if (selectingApplet()) { return; }   B1
  byte receivedByte = (byte) apdu.setIncomingAndReceive();
  ────────────────────────────────────────────────────────────
  // any code can be placed here
  DES_keys.getKey(apduBuffer, (short)0);      B2
  ────────────────────────────────────────────────────────────
  apdu.setOutgoingAndSend((short)0,16);    B3
}
```

Listing 1.2. The unwanted code

```
/*00bd*/ L0: aload_1               // apdu
/*00be*/     invokevirtual    8   // getBuffer (APDU class)
/*00c1*/     astore           4   // L4 = apduBuffer
/*00c3*/     aload_0              // this=Applet instance
/*00c4*/     invokevirtual    9   // selectingApplet()
/*00c7*/     ifeq            L1  // rel:+3 (@00CA)
/*00c9*/     return
/*00ca*/ L1: aload_1               // apdu          B1
/*00cb*/     invokevirtual    10
/*00ce*/     s2b                  // redByte
/*00cf*/     sstore           5   // L5 = redByte
─────────────────────────────────────────────────────────────
/*00d6*/     getfield_a_this 1  // DES_keys
/*00d8*/     aload            4  // L4=>apdubuffer
/*00da*/     sconst_0
/*00db*/     invokeinterface nargs : 3, index : 0, B2
                             const : 3, method: 4 //getkey
/*00e0*/     pop                 // returned Le byte
─────────────────────────────────────────────────────────────
/*00e1*/     aload_1             //L1 apdu
/*00e2*/     sconst_0
/*00e3*/     bspush 0x0F         // DES_keys size
/*00e5*/     invokeinterface  nargs : 1, index : 0, B3
                             const : 3, meth. : 1
/*00ea*/     invokevirtual    11 // setOutgoingAndSend
/*00ed*/     return
```

Listing 1.3. The virus code at the byte code level

The B1 block is the preamble, a correct code that must be executed. The B2 block corresponds to the code that must be obfuscated, and the last one B3 is the postamble. After the execution of the B1 block the state of the stack is

{ref, ref, value}. By obfuscating B2 will insert an instruction before in a such a way that constraints explained in the previous section are verified. But prior to select an instruction, we need to link statically the B2 code fragment. The final linking process is done inside the card and we can not rely on this process to resolve automatically the addresses. For that purpose, we have developed an attack, presented in [6], that provides us the way to retrieve (for most of the current cards) the linking information. For this card, the linked address of the getKey method is 0x023C. Then the code to hide becomes:

```
/*00db*/ invokeinterface nargs: 3, @023c,   method: 4
/*00e0*/ pop      // pop the return byte of the method
```

Listing 1.4. Resolved address of the B2 block

If we consider the single fault model then one of the selectable instructions is ifle (0x65) . It uses a short value and its operand is an offset to the branching instruction. The B2 code fragment to be loaded into the card is given in the listing 1.5. If the byte at the offset 0x00D6 becomes 0x0000 (thanks to the laser hit) the original B2 code will be executed.

```
/*00d6*/ [65] ifle @0x8D // 0x8D corresponds to invokestatic
/*00d8*/ [03] sconst_0   // corresponds to the nargs
/*00d9*/ [02] sconst_m1  // corresponds to the address high
/*00da*/ [3c] pop2       // corresponds to the address low
/*00db*/ [04] sconst_1   // corresponds to the method number
/*00dc*/ [3b] pop        // resynchronized with the original code
```

Listing 1.5. The hiding code

4.2   Detecting a *fault enabled virus* with SmartCM

The starting point of this study was the development of SmartCM [11], a simulator that detects such attack, and aims to analyze the effect of a fault on a Java Card program using different modules like the code mutation engine, the risk analysis tool, and the mutants reducer.

5   Conclusion

We have presented in this paper a complete CAP file engineering tool to modify each component of the CAP file in a coherent way. Within this tool, we have the possibility to design a very efficient attack using ill-typed application but also *fault enabled viruses*. It includes a stack checker to avoid embedded countermeasures and a minimalist constraint solver to generate the hiding sequence. We demonstrated the efficiency of the constraint solver to build a valid program which hides a *fault enabled virus*. We have developed a static analyzer *SmartCM* that is able to detect such a *fault enabled virus*. Recently, it appears that the

single fault model is out of date and we must consider the possibility of a dual fault attack as a valid hypothesis. Thus, the CapMap tool is able to build such a second order virus by simply applying twice the process. But the constraints for the second pass must be different, and should not reveal the hidden code. This is a new research direction on which we are working now.

References

1. Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B., Tria, A.: When Clocks Fail: On Critical Paths and Clock Faults. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 182–193. Springer, Heidelberg (2010)
2. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.-P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
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. Global Platform: Composition Model Security Guidelines for Basic Applications (2012)
6. 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. Submitted at SAR-SSI (2012)
7. Hamadouche, S.: Étude de la sécurité d'un vérifieur de Byte Code et génération de tests de vulnérabilité. Master's thesis, Université de Boumerdés (2012)
8. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Tech. rep., University of Nijmegen (2004)
9. Iguchi-Cartigny, J., Lanet, J.: Developing a trojan applets in a smart card. Journal in Computer Virology 6(4), 343–351 (2010)
10. Kömmerling, O., Kuhn, M.: Design principles for tamper-resistant smartcard processors. In: Proceedings of the USENIX Workshop on Smartcard Technology (1999)
11. Machemie, J.B., Mazin, C., Lanet, J.L., Cartigny, J.: SmartCM A Smart Card Fault Injection Simulator. In: IEEE International Workshop on Information Forensics and Security - WIFS (2011)
12. Noubissi, A., Séré, A., Iguchi-Cartigny, J., Lanet, J., Bouffard, G., Boutet, J.: Cartes à puce: Attaques et contremesures. MajecSTIC 16(1112) (November (2009)
13. Quisquater, J., Samyde, D.: Eddy current for magnetic analysis with active sensor. In: Proceedings of Esmart (2002)
14. Schmidt, J., Hutter, M.: Optical and em fault-attacks on crt-based rsa: Concrete results. In: Proceedings of the Austrochip, pp. 61–67. Citeseer (2007)
15. Skorobogatov, S., Anderson, R.: 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)
16. 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)