Countermeasures Mitigation for Designing Rich Shell Code in Java Card

Noreddine Janati, Said Elhajji, Jean-Louis Lanet

Countermeasures Mitigation for Designing Rich Shell Code in Java Card

Noreddine Janati[1], Said El Hajji[1], and Jean-Louis Lanet[2]

[1] LabMIA, Faculté des Sciences, Rabat, Morocco,
janatinoreddine@gmail.com, elhajji@fsr.ac.ma
[2] INRIA, LHS PEC
263 Avenue Général Leclerc, 35042 Rennes,
jean-louis.lanet@inria.fr
http://secinfo.msi.unilim.fr/lanet/

Abstract. Recently, logical attacks have been published that target Java based smart card. They use dynamically a type confusion which is possible if type verification is not performed. Countermeasures have been introduced on recent smart card to avoid executing rich shell code and in particular dynamic bound checking of the code segment. We propose here a new attack path for performing a type confusion that leads to a Java based self modifying code. Then, we propose to improve the previous counter measure to mitigate this new attack.

Keywords: Smart Card, Shell Code, Self Modifying Code

1 Introduction

Today most of the smart cards embed a Java Card Virtual Machine (JCVM). Java Card is a type of smart card that implements the standard Java Card 3.0 [15] 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 [12]. 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 also mixed attacks. Security issues and risks of these attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. The main assets in a smart card are the sensitive data (*i.e.* the cryptographic keys) and the code of the program. Often attackers perform cryptanalysis using side channel to recover the keys, thus breaking the confidentiality of the keys. The difficulty of breaking the security properties of these assets are in the decreasing order:

- Data confidentiality,
- Data integrity,
- Code integrity
- Code confidentiality

We have shown in the past [7], that it was relatively easy to break the code confidentiality and if we succeed then the code integrity can be broken leading to the dump of the memory. Once the memory is read it is possible to perform memory carving to gain information on the data and in particular the key containers. Smart card manufacturers increase the security of their JCVM from years to years in a such a way that published attacks do not work anymore on recent cards. The current smart cards are now well protected against pure software attacks with program counter bound checks, typed stack and so on. For such smart cards, we propose in this paper, a new attack that mitigate the secure jump countermeasure which avoid to develop rich shell code. We demonstrate first a proof of concept and then its application with the dump of a card. It is based on separating the control flow and the basic blocks of a program.

The remaining of the paper is organized as follows: the first section introduces the Java Card security. The second section presents the state of the art both in term of attacks and published countermeasures. Then, in the third section, we propose our contribution for mitigating the control flow countermeasure. Next, we propose an implementation that performs a type confusion and allows a Java based self modifying code. Finally, in the last section, we conclude.

2 Java Card Security

Smart cards security depends on the underlying hardware and the embedded software. Embedded sensors (light sensors, heat sensors, voltage sensors, *etc.*) protect the card from physical attacks. While the card detects such an attack, it has the possibility to erase quickly the content of the EEPROM preserving the confidentiality of secret data or blocking definitely the card (Card is mute). In addition to the hardware protection, softwares are designed to securely ensure that application are syntactically and semantically correct before installation and also sometimes during execution. They also manage sensitive information and ensure that the current operation is authorized before executing it.

The Byte Code Verifier (BCV) guarantees type correctness of code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in Java to perform arithmetic on reference. Thus, it must be proved that the two elements on top of the stack are of primitive types before performing any arithmetic operation. On the Java platform, byte code verification is invoked at load time by the loader. Due to the fact that Java Card does not support dynamic class loading, byte code verification is performed at loading time, *i.e.* before installing the CAP onto the card. However, most of the Java Card smart cards do not have an on-card BCV as it is quite expensive in terms of memory consumption. Thus, a trusted third party performs an off-card byte code verification and sign it, and on-card its digital signature is checked.

Moreover, the Firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets. When an applet is installed, the system uses a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are considered belonging to the same context. The firewall isolates the contexts in such a way that a method executing in one context cannot access any attribute or method of objects belonging to another context unless it explicitly exposes functionality *via* a Shareable Interface Object.

Smart card security is a complex problem with different points of view but products based on JCVM have passed successfully real-world security evaluations for major industries around the world. It is also the platform that has passed high level security evaluations for issuance by banking associations and by leading government authorities, they have also achieved compliance with FIPS 140-1 certification scheme. Nevertheless implementations have suffered severals attacks either hardware or software based. Some of them succeeded into getting access to the EEPROM (code of the downloaded applets) but as far as we know nobody succeeded into reversing the code *i.e.* having access to the code of the VM, the operating system and the cryptographic algorithm implementations. These latter are protected by the interpretation layer which denies access to other memories than the EEPROM.

3 Embedded Countermeasures

There are three main types of attacks on a smart card. The first one is the software attack [5, 9], which provides the cheapest solution to access sensitive information from the targeted cards. The second one is called side-channel or observation attack. This technique enables one either to retrieve secret cryptographic keys [8] used during a sensitive operation, or to reverse engineer the code used during a given operation [18]. The last one is the combined attack where a physical perturbation may create a logical fault which, in turn, is exploited to attack a card. We focus, in this paper, on the logical attacks which require the least knowledge for the attacker and that are the most affordable ones.

Designing a smart card attack must face several problems. The first one is the complete absence of documentation. The designer works within a black box approach. The second one is related with the embedded countermeasures. Such a product must resist to different attacks and several hardware and software fragments are dedicated to mitigate these attacks. The following section is dedicated to this class of attack and their related countermeasures.

3.1 State of the Art of Attacks against Java Cards

Logical attacks are based on the fact that the runtime relies on the 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.

Misleading the application's control flow purposes to execute a shell code stored somewhere in the memory. The aim of EMAN1 attack [14], 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. 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. Such an attack leads for the first time to execute self modifying code in a Java Card. This attack has been mitigated through different countermeasures. 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 [10] uses an underflow on the stack to get access to the return address.

When a BCV is embedded or if the process requires its usage, installing 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 `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.* [17] 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.

Hamadouche *et al.* [13] 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 shell code (virus). As they know all the addresses of each method of the `API`, they could replace instructions of any method. In [17], 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 shell code to be executed.

3.2 Mitigating the Attacks with Affordable Countermeasures

The objective of a system countermeasure is to detect an attack which occurs at linking time, run time (*e.g.* when the byte code transits on the data bus) or during the execution of another piece of code. Thus, the nature of the countermeasure is different in terms of:

- protection of variable integrity: instance field, code to be executed, evaluation stack, execution context, etc.
- protection against control flow execution modification: bypassing a test, jumping to an unauthorized data area, jumping to an argument instead of an instruction, etc.
- execution of shell code,
- type confusion, executing an instruction on an object with a given type and this object is considered in another code fragment to another type.

The integrity of application data is often used in Java Card and is called secure storage. It consists of mainly a dual storage or a checksum in order to verify whether the modification of the field is only done through the virtual machine (VM). Another integrity check concerns the VM structure and in particular the frame context. Using the EMAN 2 attack it is possible to modify the return address in the frame using unchecked local variable indexes. Most of smart cards available on the web markets might be flooded by the modification of the `CFG`. Thus, it is possible to jump into an array which contains any shell code.

For preventing the execution of a shell code, there is the possibility to re-encode on the fly during the linking phase of the value of byte code. So, if someone trying to execute an arbitrary array will not be able to obtain the desired behavior. In such a method the encoded value depends on a dynamic variable, using the `JPC` for example as a nonce is enough to avoid any brute force attack for guessing the scrambled value.

There are lot of possibilities to protect the data and the execution of a code into the VM. Unfortunately, if all of them are activated during the execution of an application, the performance of the smart card will drastically decrease reaching an unacceptable level. For that reason, most of the smart cards available on web market implement the bound check counter measure which has been demonstrated as efficient enough to mitigate any exploitable shell code.

3.3 Checking the Jump Boundaries

An affordable countermeasure against the execution of shell code is to verify if the code is still executing within the boundaries of the current method. For each method, the system maintains several information like `maxJPC`. So, the domain of the `JPC` of a method belongs to {0..`maxJPC`}. An attack like the EMAN2

presented in the previous section, modifies the return address such that as it returns from method `f()` the control is transferred to the shell code instead of the caller. But the execution of the shell code is done within the execution context of the caller as shown Figure 1. In such a case, when the shell code ends with its own `return` instruction, it goes back to the caller of the caller of the method `f()`. The shell code can not be embedded within the method `f()` and thus is implemented as an array stored in a different area of the method.

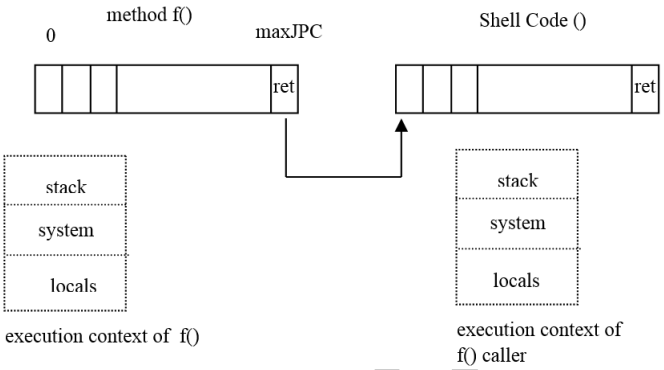

Fig. 1. Description of execution context

A naive solution should be to check if the value of `JPC` belongs to its domain as shown in the code fragment of the Listing 1.1.

Listing 1.1. Check the boundaries for each instruction

```
int16 BC_pop(void)
{
    vm_sp--;
    vm_pc += 1;
    if (vm_pc > maxJPC)
        return BC_SECURITY();
    return ACTION_NONE;
}
```

This increases the size of each byte code of 16 bytes on an ARM7 architecture. The original instruction requires 44 bytes. The increase for each instruction is around 36% which is too much for such a small device. The trade off is to check only the jump destination while the control flow is transferred. Thus, only the exit of a basic block will be checked, reducing the overhead. The exit instructions belong to the set {`if_xx_yy`, `goto_yy`} with `yy` having the value `wide` or not, depending of the domain of the offset coded on a byte or a short. The term `xx` has the values `type`,`ne, eq, lt, ge, gt`. The overhead is drastically lower

impacting only a subset of the instruction set. On the Oracle Purse application it represents only 4% of overhead on the same architecture. This countermeasure is affordable and is able to detect that the control flow has been transferred to a shell code if this one requires a branch.

It does not prevent to jump to a shell code but restrict the semantics of the shell code to a linear code. In particular no loop is available, no condition evaluation and so on. As an effect, it becomes impossible to use a shell code for dumping the memory.

4 Mitigating the Control Flow Countermeasures

Two solutions are possible to bypass the countermeasure. Both of them are related to the non completeness of the countermeasure. The first one is to use the exception mechanism to transfer the control flow and data to the caller. It requires in the caller to rebuilt the control flow using the catch mechanism of Java. Thus, the exception object is propagated to the caller if an handler is present it can take decision using the `reason` embedded in the exception object. The second possibility is to simply use the return mechanism of Java if correctly handled. We have chosen the second but any avatar using the exception can be used with the same result.

The first step consist in implementing an EMAN2 as described by Bouffard *et al.* [5]. This attack abuses the instructions that access the local stack area[3] in order to write outside the domain of the locals. The authors succeeded in modifying the return address. When the `return` instruction is executed, this leads to a controlled execution flow modification.

A fragment of the EMAN2 exploit is shown in the Listing 1.2. The described function contains two parameters (the class instance, `this`, and the `address` parameter) and no local variable. In this function, the `sload 1` operation pushes the value of `address` parameter onto the Java Card stack. The following operation, `sstore 4`, stores the last pushed short value into the local variable 4.

Listing 1.2. Stack overflow into a Java Card.

```
public void updateReturnAddress (short address) {
02 // flags:0 max_stack:2
20 // nargs:2 max_locals:0
16 01  sload  1 // push address from the local 1
29 03  sstore 4 // STACK OVERFLOW!
7A     return   // Jump to the shellcode
}
```

[3] As defined in the Java Card specification [15], accessing to the local variable is done by the `aload`, `astore`, `sload` and `sstore` instructions.

As the function's stack contains only two elements into the locals part, the authors made a stack overflow from the local variable area to set up the return address[4] by a specific value. The state of the Java Card stack is presented into the Figure 2 at left. For the current frame, we find first the arguments of the method and then the locals variables. Often, a system area is used to be able to retrieve the state of the caller. Above that is the evaluation stack. We have found some cards where the system area is not contiguous with the locals and the stack as shown in the Figure 2 right.

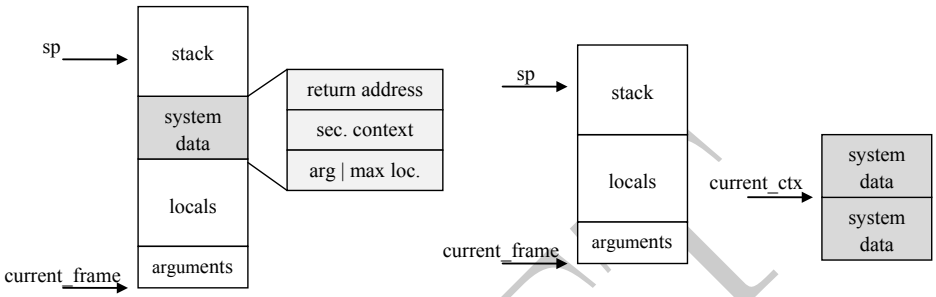

Fig. 2. Stack

4.1 Principle of the Control Flow Extraction

The objective of the attack is to split the original code fragment that we want to execute even in presence of the countermeasure into several basic blocks. Then, an instruction `sspush value` is inserted and the value is the variable that is evaluated at the beginning of the next basic block. An instruction `sreturn` finishes each of the basic block. All these basic blocks are stored consecutively into an array. The control flow is then assumed by a specific method `controlFlow()`. The `CFG` is implemented into this method which contains only decision and call to the `dummy()` method. This method plays only the role to be the context execution of the shell code and just invoke the method `shellCodeLauncher()`. This latter is the one patched thanks to the EMAN2 attack.

Once the `shellCodeLauncher()` ends its execution it transfers the control flow to one of the basic block stored into the array. At the end of the shell code the `return` instruction is executed which transfer the control flow to the method `controlFlow()` as shown in the Figure 3. It is important to notice that the execution context of the shell code is the `dummy` method and not the method `shellCodeLauncher()`.

With such an architecture, illegal code is executed in the `shellCode` method while the `CFG` is managed by the `controlFlow` method.

[4] On the evaluated smart cards, the references are encoded on 2-byte as short values.

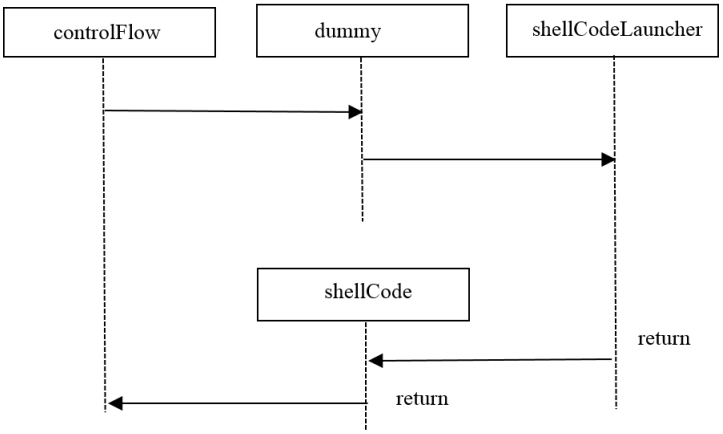

Fig. 3. Control flow derivation

4.2 Parameters Exchange between the Controller and the Shell Code

We have seen how retrieving data from the shell code using simply the value pushed on top of the stack and send back to the caller. To provide input data to any of the basic blocks stored into the array, we can use the caller context *i.e.* the argument of the `dummy` method. The number of argument of the dummy method must be the max argument of all the basic blocks for each type of data.

For example, if the shell code is made of three basic blocks requiring the following data: $BB_1 = short, byte, BB_2 = byte, byte, BB_3 = ref, short, byte$ the maximum of generic argument of `dummy` method is 4 defined as $L_0 = short, L_1 = byte, L_2 = byte, L_3 = ref$. Note that BB_3 will be called with a reordering of its arguments: $BB_3 = short, byte, ref$. Then the argument used by each basic blocks will be the following $BB_1 = L_0, L_1, BB_2 = L_1, L_2, BB_3 = L_0, L_1, L_3$. For each basic blocks, the unused variables are set to their default Java value.

The first parameter of the dummy method is the offset to jump into the shell code array. In the code fragment given in the Listing 1.3, the first call in the evaluation condition is for the first segment of the shell code with the related parameters, the size of the first segment (n) is then added at the first parameter of the second second segment leading to a call to BB_2. The size of the first and the second (m) is then used to call the third segment.

Listing 1.3. Calling convention of the basic blocks

```
...
if (dummy (arAdd,L_0,L_1,0,null)) // implicit call to BB1
 dummy (arAdd+n,L_0,L_1,L_2,null); //implicit call to BB2
else
 dummy (arAdd+m,L_0,0,L_2,L3); //implicit call to BB3
}
```

The only constraint is that the order of the parameters of the `dummy` method must be strictly the same than the method `shellCodeLaucher`.

5 Experiments: the Java Self Modifying Code Revisited

We use our method to execute polymorphic code, *i.e.* a code that modifies itself like a virus to be able to execute illegal instruction. This shell code is able to dump completely the memory of the card even in presence of the countermeasure.

5.1 Type Confusion Exploitation

The idea is to use in the `controlFlow` method an array that can be manipulated with read and write instructions and the shell code that execute the array. In the shell code, we use the instruction `getstatic_s` that retrieves the value of a short at the given index as shown in 1.4. The value of the index is an argument of the instruction and cannot be incremented directly by the method. The parameter of the instruction is an index in the constant pool before the link resolve the token and becomes inside the card an offset, or reference depending to the implementation.

Listing 1.4. Simple shell code to dump the memory

```
7  public void getMyAddress(){
8    // flags: 0  max_stack : 1
9    // nargs: 0  max_locals: 0
10   7D 00 00  getstatic_s 2
11   78        sreturn
12 }
```

The corresponding value in the shell code array is [7D, 00, 00, 78]. Executing this shell code retrieves the content of the memory at the address 0x0000. The `controlFlow` method has to manage the value of the address. In this basic example, the input data are only the offset in the array and the return value of the basic block must be stored in input-output buffer to be send to the reader. The address to be modified is the content of the shell code array at offset 1 for the high byte of the address and 2 for the low one. The aim is to write in the input output buffer 128 bytes of memory.

Listing 1.5. Calling the shell code with parameters and retrieving return value

```
1  public void controlFlow(APDU apdu, byte[] buffer, short arAdd){
2   short boff=0x00;
3   for(short i=0;i<=0x7F; i++){
4   short x=dummy (arAdd);
5   Util.setShort(buffer, boff, x);
6    if(shellcode[2]==(byte)0xFF){
7      shellcode[2]=(byte)0x00;
8      shellcode[1]++);}
```

```
9     else{shellcode[2]+=2;}
10      boff=(short)(boff+2);
11    }
12   apdu.setOutgoingAndSend((short) 0x00,boff);
13  }
```

In the Listing 1.5 of the `controlFlow` method at line 4 we get the content of the memory and line 5 we store it in the buffer. Line 9 we increase the value of the address to be dumped, and from line 6 to 8 we propagate the carry.

5.2 Completeness of the Countermeasure

We have demonstrated that such a counter measure is inefficient due to its incompleteness. The objective of the initial countermeasure was to detect the execution of a shell code outside its original position by checking the destination branch. Thus the current counter measure encompass only the set of intra procedure instructions (*i.e.* `goto, if, jsr`). It must be extended to the set of intra procedure instructions which is more complicated. The VM has the information about the `minJPC` and the `maxJPC` which is enough to check destination branch within the boundaries.

For intra procedure instructions the VM needs to know while building or destroying the frame if the `JPC` belongs to a valid method. A valid method `JPC` depends how methods are stored within the class. One can suggested to define the boundaries of the method pool but if the method is inherited, the check must be done with the mother class and not the current one. Moreover the method must be allowed to be called according to the current instance. This is threaten naturally by the `invoke` instruction while building the frame, no new check is required. The `return` instruction is more difficult to handle but one invariant at least must hold: at the destination the previous instruction must be an `invoke` instruction.

Listing 1.6. Deleting the frame

```
1  bool releaseFrame(value_t *ret_val)
2  {   /* mark this frame as free */
3      thr_active->curr_frame->method = NULL;
4      if (thr_active->curr_frame->prev == NULL)
5          return false;
6      /* update link pointers */
7      thr_active->curr_frame = thr_active->curr_frame->prev;
8      thr_active->curr_frame->next = NULL;
9      /* copy return value in case it exists */
10     *ret_val = *(--vm_sp);
11     /* update SP and PC */
12     vm_sp = thr_active->curr_frame->sp;
13     if (thr_active->curr_frame->pc - 3 == BC_invoke){
14       vm_pc = thr_active->curr_frame->pc;
```

```
15          return true;}
16        else return false;
17  }
```

The check of the invariant can be done by the method that restores the previous frame as shown in Listing 1.6. We added a verification line 13 if the previous instruction was the generic `invoke` it returns true else false and the caller must handle the security problem. The overhead occurs only while retrieving the previous instruction and it ensures the completeness of the countermeasure.

6 Conclusion and Future Works

In this paper we have demonstrated that a well known countermeasure against shell code execution can be bypassed if not all the instructions are covered by the dynamic checks. We have shown the possibility to extract the control flow and to generate a shell code that corresponds any executable program. We use the method parameter *i.e.* its signature to provide input and recover data from the shell code. The control program can use a type confusion to execute a rich shell code, using self modifying code. As a proof of concept, we developed a program with its controller where the controller fills an array that is executed by the shell code. We have been able to dump entirely the memory.

We develop now a program to automatically extract the controller and the shell code for any program. Then, we expect to be able to reverse the content of the dumped memory thanks to a memory carving program under development.

References

1. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Prouff [16], pp. 297–313
2. Barbu, G., Giraud, C., Guerin, V.: Embedded Eavesdropping on Java Card. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC. IFIP Advances in Information and Communication Technology, vol. 376, pp. 37–48. Springer (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. Lecture Notes in Computer Science, vol. 7159, pp. 1–13. Springer (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. Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer (2010)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: CARDIS, pp. 283–296 (2011)
6. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff [16], pp. 283–296
7. Bouffard, G., Lanet, J.L.: Reversing the operating system of a java based smart card. Journal of Computer Virology and Hacking Techniques 10(4), 239–253 (2014), http://dx.doi.org/10.1007/s11416-014-0218-7

8. Carlier, V., Chabanne, H., Dottax, E., Pelletier, H.: Electromagnetic Side Channels of an FPGA Implementation of AES. IACR Cryptology ePrint Archive 2004, 145 (2004)
9. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon and Rohatgi [11], pp. 140–151
10. Faugeron, E.: Manipulating the Frame Information With an Underflow Attack. In: Francillon and Rohatgi [11]
11. Francillon, A., Rohatgi, P. (eds.): Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers, Lecture Notes in Computer Science, vol. 8419. Springer (2014)
12. GlobalPlatform: Card Specification. GlobalPlatform Inc., 2.2.1 edn. (Jan 2011)
13. Hamadouche, S., Lanet, J.L.: Virus in a smart card: Myth or reality ? In: Cheng, L., Wong, K. (eds.) Journal of Information Security and Applications, vol. 18 issues 2-3, pp. 130–137. Elsevier (2013)
14. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6(4), 343–351 (2010)
15. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. No. Version 3.0.4, Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2011)
16. Prouff, E. (ed.): Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7079. Springer (2011)
17. 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., García-Alfaro, J. (eds.) DBSec. Lecture Notes in Computer Science, vol. 7371, pp. 122–128. Springer (2012)
18. Vermoen, D., Witteman, M.F., Gaydadjiev, G.: Reverse Engineering Java Card Applets Using Power Analysis. In: WISTP. pp. 138–149 (2007)