A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks

Guillaume BOUFFARD

PhD Defenced the 10th of October, 2014
Supervised by Pr. Jean-Louis Lanet
guillaume.bouffard@ssi.gouv.fr

Prix de thèse CNRS 2015
Action: « Objects intelligents Sécurisés et Internet des Objets »
June 17th, 2015
Outline

Introduction
  Smart Card
  Java Card Technology

Contribution
  Fault Tree Analysis
  Smart Card Vulnerability Analysis using Fault Tree Analysis
  Corrupting the Java Card’s Control Flow
  Security Automatons to Protect the Java Card Control Flow

Conclusion
The Smart Card

- Used in our everyday life:
  - Credit Card;
  - (U)SIM Card;
  - Health Card (French Vitale card);
  - Pay TV;
  - ...

- Tamper-Resistant Computer;
- Securely stores and processes information;
- Most of the smart cards are based on Java Card technology.

This device contains sensitive data
Java Card Security Model

- **Off-card security**
  - Java CLASS Files → Byte Code Converter → Byte Code Verifier (BCV) → Byte Code Signer → Java Card Files

- **On-card security**
  - Java Card Files → Embedded Lightweight BCV → Installed applet → Firewall
Java Card Attacks

Physical attacks

▶ Side Channel attacks (timing attacks, power analysis attack, etc.);
▶ Fault attacks (electromagnetic injection, laser beam injection, etc.).

Logical attacks

▶ Execution of malicious Java Card byte codes.

Combined attacks

▶ Mix of physical and logical attacks.
Problematic

- Inductive Approach:
  - 1 attack = 1 countermeasure;
  - **Bottom-up** approach.
Problematic

- **Inductive Approach:**
  - 1 attack = 1 countermeasure;
  - **Bottom-up** approach.

- **Thesis Objectives:**
  - Find and prevent each undesirable events;
  - Global vision to protect the smart card’s assets;
  - Design a *top-down analytic approach*. 
The Fault Tree Analysis (FTA)

- Undesirable events;
- Initial causes;
- Gate connectors.

Diagram:
- Root undesirable event
  - Effect 1
  - Intermediate undesirable event
    - Effect 2
    - Effect 3
  - E1
The Fault Tree Analysis (FTA)

- Undesirable events;
- Initial causes;
- Gate connectors.
The Fault Tree Analysis (FTA)

- Root undesirable event
  - Effect 1
  - Intermediate undesirable event
    - Effect 2
    - Effect 3

- Undesirable events;
- Initial causes;
- Gate connectors.
The Fault Tree Analysis (FTA)

- Undesirable events;
- Initial causes;
- Gate connectors.
Smart Card’s Assets

- The smart card’s assets are the **code** and the **data**;

- Security properties:
  - Integrity;
  - Confidentiality;

- Undesirable events can affect:
  - Code integrity;
  - Data integrity;
  - Code confidentiality;
  - Data confidentiality;
Smart Card’s Assets

- The smart card’s assets are the **code** and the data;
- Security properties:
  - Integrity;
  - Confidentiality;
- Undesirable events can affect:
  - **Code integrity**;
  - Data integrity;
  - Code confidentiality;
  - Data confidentiality;

*An attack offers the execution of a malicious byte code.*
Execution of a malicious code

Frame Corruption
- Return address modification
- Context corruption
- Confusing invoker's state

Code desynchronisation
- Corrupting the branching instructions
- Faulty table jumping operations

Control flow corruption
- Corrupting finally-clause
- Fooling the exception mechanism
- Invoking an unexpected function
- Type confusion

For this presentation, two vulnerabilities will be introduced:
- Modifying the return address;
- Corrupting the finally-clause.

Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Execution of a malicious code

- Execution of a malicious code
  - Frame Corruption
    - Return address modification
    - Context corruption
    - Confusing invoker's state
  - Code desynchronisation
    - Corrupting the branching instructions
  - Control flow corruption
    - Faulty table jumping operations
    - Corrupting finally-clause
    - Fooling the exception mechanism
    - Invoking an unexpected function
    - Type confusion

- For this presentation, **two vulnerabilities** will be introduced:
  - Modifying the return address;
  - Corrupting the finally-clause.

- Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Execution of a malicious code

For this presentation, **two vulnerabilities** will be introduced:

- **Modifying the return address**;
- **Corrupting the finally-clause**.

Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Execution of a malicious code

For this presentation, **two vulnerabilities** will be introduced:

- Modifying the return address;
- **Corrupting the finally-clause.**

Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Execution of a malicious code

- Execution of a malicious code
- Security Automatons

- Frame Corruption
  - Return address modification
  - Context corruption
  - Confusing invoker’s state

- Code desynchronisation
  - Corrupting the branching instructions
  - Faulty table jumping operations
  - Corrupting finally-clause

- Control flow corruption
  - Fooling the exception mechanism
  - Invoking an unexpected function
  - Type confusion

- Finally-clause

For this presentation, two vulnerabilities will be introduced:
- Modifying the return address;
- Corrupting the finally-clause.

Thanks to **minimal cut set**, a countermeasure to protect the execution flow was developed: the **security automatons**.
The Java Method Return

The current frame is used in this case to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method’s frame with the returned value (if any) pushed onto the operand stack of that frame.  (source: Java 8 Virtual Machine Specification)

▶ A frame header may include:

- Previous frame’s size;
- Program counter of the invoker (the return address);
- Security context of the invoker.
The current frame is used in this case to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method’s frame with the returned value (if any) pushed onto the operand stack of that frame. (source: Java 8 Virtual Machine Specification)

A frame header may include:

- Previous frame’s size;
- Program counter of the invoker (the return address);
- Security context of the invoker.
EMAN2: A Ghost In the Stack

- Modifying the return address;

Return address modification

Override the return address

- Overflow from the local variable [Bouffard et al., CARDIS 2011]
- Underflow from the operand stack [Faugeron, CARDIS 2013]

- No BCV

 ill-formed code

- Code modification
  - No frame check
  - Fault Injection
EMAN2 and Its Avatars

- **Stack overflow** from the local variables [Bouffard et al., CARDIS 2011]
  - sstore, sinc, etc.;

- **Stack underflow** from the operand stack [Faugeron, CARDIS 2013]
  - dup_x, swap_x, etc.;

- Countermeasures from the literature:
  - Checking the integrity of the frame’s header data;
  - Verifying each access to the frame’s areas [Lackner et al., CARDIS 2012];
  - Scrambling the memory [Barbu’s PhD Thesis, 2012] [Razafindralambo et al., SNDS 2012];
  - These countermeasures are at the same level than the attacks which they prevent;
EMAN2 and Its Avatars

- Stack **overflow** from the local variables [Bouffard et al., CARDIS 2011]
  - sstore, sinc, etc.;

- Stack **underflow** from the operand stack [Faugeron, CARDIS 2013]
  - dup_x, swap_x, etc.;

- Countermeasures from the literature:
  - Checking the integrity of the frame’s header data;
  - Verifying each access to the frame’s areas [Lackner et al., CARDIS 2012];
  - Scrambling the memory [Barbu’s PhD Thesis, 2012] [Razafindralambo et al., SNDS 2012];
  - These countermeasures are at the same level than the attacks which they prevent;

- This attack modifies the Java Program Counter value upon the return address register. Recent smart cards embed countermeasures against this attack! ... **only the path is protected**;
The **finally-Clause**

- A finally-statement used the `jsr` ("*jump to subroutine*") and `ret` ("*return from subroutine*") instructions (deprecated since Java 6);
- The `jsr` pushes the address of the instruction immediately following it (typed as `ReturnValueAddress`);
- Saves the return value (if any) in a local variable;
- The `ret` instruction continues the execution from the value saved in the local variable.
Corrupting the finally-Clause

- Setting a creepy ReturnAddress
- Malicious code
  - No BCV
  - Code modification
    - Type confusion
      - No typed stack
    - Fault Injection
      - No typed heap

[Bouffard et al., CARDIS 2014]
How to Exploit the \texttt{jsr} instruction?

\begin{itemize}
\item \textbf{Hypothesis:}
\begin{itemize}
  \item No verified by a BCV
  \item No typed stack
\end{itemize}
\end{itemize}

\begin{verbatim}
short jsrAttack () {
  01 // flags: 0 max_stack : 1
  11 // nargs: 1 max_locals: 1
  /*0x53*/ L0: jsr L2
  /*0x56*/ L1: sspush 0xCAFE
  /*0x59*/ sreturn
  /*0x5A*/ sspush 0xBEEF
  /*0x5D*/ sreturn
  /*0x5E*/ L2: astore_1
  /*0x5F*/ sinc 0x1, 0x4
  /*0x62*/ ret 1 // -> L1
}
\end{verbatim}
How to Exploit the jsr instruction?

▶ Hypothesis:
- No verified by a BCV
- No typed stack

```java
short jsrAttack () {
    01 // flags: 0 max_stack : 1
    11 // nargs: 1 max_locals: 1
    /*0x53*/ L0: jsr L2
    /*0x56*/ L1: sspush 0xCAFE
    /*0x59*/ sreturn
    /*0x5A*/ sspush 0xBEEF
    /*0x5D*/ sreturn
    /*0x5E*/ L2: astore_1
    /*0x5F*/ sinc 0x1, 0x4 Type confusion
    /*0x62*/ ret 1 // -> L1
}
```
Cheating the BCV component

- The BCV checks the structure and the semantics of the application;
- To verify the byte code semantics, the BCV starts its analyse from an entry point;
- Unreachable code has no entry point $\Rightarrow$ it is not checked by the BCV!
- A malicious byte code can be hidden through the BCV verification!
An Unreachable Code...

```java
void cheatingBCV () {
    04 // flags: 0 max_stack : 4
    03 // nargs: 0 max_locals: 3
    /*0x05B*/ L0: jsr L1
    // ...
    /*0x066*/ L1: astore_3
        L2: ... // Set of instructions
    /*0x163*/ if_scmpeq_w 0xFF05 // -> L2
    /*0x166*/ return
    /*0x167*/ sinc 0x3, 0x4
    /*0x16A*/ ret 0x3
}
```
An Unreachable Code...

```java
void cheatingBCV () {
    // flags: 0 max_stack : 4
    // nargs: 0 max_locals: 3

    /*0x05B*/ L0: jsr L1
    // ...
    /*0x066*/ L1: astore_3
    L2: ... // Set of instructions
    /*0x163*/ if_scmpeq_w 0xFF05 // -> L2
    /*0x166*/ return
    /*0x167*/ sinc 0x3, 0x4
    /*0x16A*/ ret 0x3
}
```

Checked by the BCV

Unchecked by the BCV
void cheatingBCV () {
    // flags: 0 max_stack : 4
    // nargs: 0 max_locals: 3
    /*0x05B*/ L0: jsr L1
    // ...
    /*0x066*/ L1: astore_3
    L2: ... // Set of instructions
    /*0x163*/ if_scmpeq_w 0xFF05 // -> L2
    /*0x166*/ return
    /*0x167*/ sinc 0x3, 0x4
    /*0x16A*/ ret 0x3
}

Checked by the BCV
Unchecked by the BCV

verifycap api_export_files/**/*.exp maliciousCAPFile.cap
[ INFO: ] Verifier [v3.0.5]
[ INFO: ] Copyright (c) 2015, Oracle and/or its affiliates.
    All rights reserved.

[ INFO: ] Verifying CAP file maliciousCAPFile.cap
[ INFO: ] Verification completed with 0 warnings and 0 errors.
EMAN4 [Bouffard et al., CARDIS 2011] introduced a way to change an instruction’s parameter upon a laser beam injection;

- This attack focuses on wide instructions;
- goto_w, if_*_w, ...
EMAN4 [Bouffard et al., CARDIS 2011] introduced a way to change an instruction’s parameter upon a laser beam injection;

- This attack focuses on wide instructions;
- `goto_w, if_*_w, ...`

- `if_scmpeq_w 0xFF05`
EMAN4 [Bouffard et al., CARDIS 2011] introduced a way to change an instruction’s parameter upon a laser beam injection;
  ◦ This attack focuses on wide instructions;
  ◦ goto_w, if_*_w, ...

if_scmpeq_w 0xFF05 \Rightarrow if_scmpeq_w 0x0005
EMAN4 [Bouffard et al., CARDIS 2011] introduced a way to change an instruction’s parameter upon a laser beam injection;

- This attack focuses on wide instructions;
- goto_w, if_*_w, ...

- if_scmpeq_w 0xFF05 \Rightarrow if_scmpeq_w 0x0005
- That can be viewed as a logical attack enabler.
Protecting the Execution Flow

▶ Direct modification:
  ○ Integrity → can be bypassed when the JPC is updated by the JCVM;

▶ Transient fault:
  ○ Executing twice the same piece of code;
  ○ It is a very expensive solution;

▶ Solution: dynamically check the applet’s CFG:
  ○ Séré’s countermeasures [Séré’s PhD thesis, 2010] based on Field of bits, Basic block method or Path check technique;
  ○ This kind of countermeasure can be **computed in the card**?
Security Automatons and Execution Monitor

Principle

- Detecting a deviant behaviour $\Rightarrow$ \textbf{safety property} “\textit{nothing bad happens}”; 
- Preventing some attacks: several partial traces of events are defined: 
  - Property can be encoded by a finite state automaton; 
- Schneider automatons: $(Q, q_0, \delta)$, where $Q$ is a set of states, $q_0$ is the initial state and $\delta$ is a transition function $(Q \cdot I) \rightarrow 2^Q$; 
- The CFG can be computed during the loading process; 
- When interpreting a byte code, the monitor checks: 
  - If the transition generates an authorized partial trace; 
  - If not, it takes an appropriate countermeasure.
Security Automatons and Execution Monitor (Cont.)

Principle

Security automaton
(computed inside the card)

- [Bouffard et al., SSCC 2013], [Bouffard et al., SAR-SSI 2013] and extended in [Bouffard et al., IJTMCC 2014].

State matrix (binary implementation of the security automaton)

<table>
<thead>
<tr>
<th>State</th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td>$\delta_1$</td>
<td>$\delta_3$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$\delta_5$</td>
<td></td>
<td>$\delta_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\delta_4$</td>
<td>$\delta_5$</td>
<td></td>
</tr>
</tbody>
</table>
Security Automaton in Practice

```java
protected ProtocolPayment (byte[] buffer, short offset, byte length) {
    A[0] = 0; // initialisation of array A
    for (byte j = 0; j < buffer[(byte)(offset+12)]; j++) {
        D[j] = 0; // initialisation of array D
    }
    pin = new OwnerPIN((byte) TRY_LIMIT, (byte) MAX_PIN_SIZE);
    // Initialisation of pin
    pin.update(myPin, (short) START_OFFSET, (byte) myPin.length);
    register(); // registering this instance
} // source: (Girard et al., CRiSIS 2010)
```
Security Automaton in Practice

```java
protected ProtocolPayment (byte[] buffer, short offset, byte length) {
    A[0] = 0; // initialisation of array A
    for (byte j = 0; j < buffer[(byte)(offset+12)]; j++) {
        D[j] = 0; // initialisation of array D
    }
    pin = new OwnerPIN((byte) TRY_LIMIT, (byte) MAX_PIN_SIZE);
    // Initialisation of pin
    pin.update(myPin, (short) START_OFFSET, (byte) myPin.length);
    register(); // registering this instance
} // source: (Girard et al., CRiSIS 2010)
```

To create the security automaton:

- Local view of the method’s CFG;
- The set $S$ contains the element of a language which expresses the control flow integrity policy:
  - ifeq, ifne, goto, invoke, return, etc.;
  - plus the dummy instruction `join` representing any other instruction pointed by a label.
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2

/*0x3A*/ L2: sload_3
... 
/*0x42*/ if_scmplt L1

/*0x30*/ L1: getfield_a_this 1
... 
/*0x39*/ sstore_3

/*0x4A*/ L3: aload_0
... 
/*0x4B*/ invokespecial 5
... 
/*0x56*/ invokevirtual 7
... 
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```
Security Automaton included in the JCVM

/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0
... invokespecial 5
/*0x56*/ invokevirtual 7
... invokevirtual 8
/*0x5D*/ return

The trace recognised would be:
(goto, (if_scmplt, join), !if_scmplt, return)
Security Automaton included in the JCVM

```java
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
...*/0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
...*/0x39*/ sstore_3
/*0x4A*/ L3: aload_0
...*/0x4B*/ invokespecial 5
...*/0x56*/ invokevirtual 7
...*/0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:
- `goto`, `if_scmplt`, `join`, `!if_scmplt`, `return`
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2

/*0x3A*/ L2: sload_3
...  
/*0x42*/ if_scmplt L1

/*0x30*/ L1: getfield_a_this 1
...  
/*0x39*/ sstore_3

/*0x4A*/ L3: aload_0
...  
/*0x4B*/ invokespecial 5
...  
/*0x56*/ invokevirtual 7
...  
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:

```
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

Goto L2

If_scmplt L1

Start → 0 goto 2 !if_scmplt 3

If_scmplt

Join

/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3...
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1...
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0...
/*0x4B*/ invokespecial 5...
/*0x56*/ invokevirtual 7...
/*0x5A*/ invokevirtual 8
/*0x5D*/ return

The trace recognised would be:
(goto, (if_scmplt, join), !if_scmplt, return)
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2

/*0x3A*/ L2: sload_3
...
/*0x42*/ if_scmplt L1

/*0x30*/ L1: getfield_a_this 1
...
/*0x39*/ sstore_3

/*0x4A*/ L3: aload_0
...
/*0x4B*/ invokevirtual 5
...
/*0x56*/ invokevirtual 7
...
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:

```
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

```java
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3...
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1...
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0...
/*0x4B*/ invokespecial 5...
/*0x56*/ invokevirtual 7...
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:

```plaintext
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2

/*0x3A*/ L2: sload_3
...
/*0x42*/ if_scmplt L1

/*0x30*/ L1: getfield_a_this 1
...
/*0x39*/ sstore_3

/*0x4A*/ L3: aload_0
...
/*0x4B*/ invokespecial 5
...
/*0x56*/ invokevirtual 7
...
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:
```
(goto, (if_scmplt, join), !if_scmplt, return)
```

Guillaume BOUFFARD  A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks

23/30
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
...  
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
...  
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0
...  
/*0x4B*/ invokespecial 5
...  
/*0x56*/ invokevirtual 7
...  
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:

```
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

The trace recognised would be:

(start) goto (if_scmplt), (join) !if_scmplt, return
Security Automaton included in the JCVM

Start:

0: goto 2

2: !if_scmplt 1

1: if_scmplt

1: join

2: goto 3

3: return

0: goto 2

2: !if_scmplt

/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
/*0x39*/ sstore_3
/*0x3A*/ L2: sload_3
/*0x4A*/ L3: aload_0
... invokespecial 5
/*0x56*/ invokevirtual 7
... invokevirtual 8
/*0x5D*/ return

The trace recognised would be:

\((\text{goto} , (\text{if} \_\text{scmplt} , \text{join}) ) , \text{!if} \_\text{scmplt} , \text{return})\)
Security Automaton included in the JCVM

```java
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0
/*0x4B*/ invokespecial 5
/*0x56*/ invokevirtual 7
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:
```
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

start → 0

goto → 1

!if_scmplt → 2

if_scmplt → 1

join → 3

/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
... /*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
... /*0x39*/ sstore_3
/*0x4A*/ L3: aload_0
... /*0x4B*/ invokespecial 5
... /*0x56*/ invokevirtual 7
... /*0x5A*/ invokevirtual 8
/*0x5D*/ return

The trace recognised would be:
(goto, (if_scmplt, join), !if_scmplt, return)
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0
/*0x4B*/ invokespecial 5
/*0x56*/ invokevirtual 7
/*0x5A*/ invokevirtual 8
/*0x5D*/ return
```

The trace recognised would be:
```
(goto, (if_scmplt, join), !if_scmplt, return)
```
Security Automaton included in the JCVM

```
/*0x03*/ L0: aload_0
/*0x04*/ invokespecial 6
/*0x2E*/ goto L2
/*0x3A*/ L2: sload_3 ...
/*0x42*/ if_scmplt L1
/*0x30*/ L1: getfield_a_this 1 ...
/*0x39*/ sstore_3
/*0x4A*/ L3: aload_0 ...
/*0x4B*/ invokespecial 5 ...
/*0x56*/ invokevirtual 7 ...
/*0x5A*/ invokevirtual 8 /*0x5D*/ return
```
Security Automaton included in the JCVM

The trace recognised would be:
(goto, (if_scmplt, join)*, !if_scmplt, return)
Security Automaton included in the JCVM (Cont.)

\begin{center}
\begin{tikzpicture}
\node[state] (0) at (0,0) {0};
\node[state] (1) at (1,1) {1};
\node[state] (2) at (2,2) {2};
\node[state] (3) at (1,3) {3};
\node[state] (4) at (3,2) {4};
\node[state] (5) at (2,3) {5};
\node[state] (6) at (3,1) {6};
\node[state] (7) at (4,2) {7};
\node[state] (8) at (5,1) {8};

\path[->] (0) edge node {start} (1);
(1) edge node {invokepecial 6} (2);
(2) edge node {if_scmplt} (3);
(3) edge node {if_scmplt} (1);
(2) edge node {goto} (4);
(4) edge node {!if_scmplt} (5);
(5) edge node {join} (6);
(6) edge node {invokepecial 5} (7);
(7) edge node {invokevirtual 8} (8);
(7) edge [loop above] node {return} (7);
\end{tikzpicture}
\end{center}
Security Automaton included in the JCVM (Cont.)

<table>
<thead>
<tr>
<th>δ</th>
<th>q0</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
<th>q6</th>
<th>q7</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokespecial 6 goto join if_scmplt invokespecial 5 invokevirtual 7 invokevirtual 8 return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>q1</td>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>q2</td>
<td></td>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3,4</td>
<td></td>
<td>q3,4</td>
<td></td>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td></td>
<td></td>
<td>q5</td>
<td></td>
<td></td>
<td>q6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td></td>
<td></td>
<td></td>
<td>q6</td>
<td></td>
<td></td>
<td>q7</td>
<td>+</td>
</tr>
</tbody>
</table>

Guillaume BOUFFARD  A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks
The Security Automaton

- The execution flow is checked by the security automaton upon a finite state machine;
- Each transition is verified by the execution monitor;
- The CFG can be automatically computed by the loading process;
- The CFG can be encoded upon a sparse matrix → optimised solution to store the CFG
- The JCVM and the loader should be modified to handle automatons.
Conclusion

- This thesis aimed at designing efficient and affordable countermeasure using a top-down approach;
- It is based on the Fault Tree Analysis which this approach aims at being generic;
- We identified major undesirable events:
  - We discovered new attack paths, someones are generic;
  - And introduced high level-countermeasures.
Conclusion (Cont.)

- We focused on the **code integrity**:  
  - Modification of the control flow;  
  - Corruption of the Java Card Linker [Hamadouche et al., SAR-SSI 2012], [Razafindralambo et al., SNDS 2012] and [Bouffard et al., CRiSIS 2013];

- Each evaluated attacks succeeded on different cards  
  - Bottom-up approach ?  
  - We wear a white hat;

- Our approach aims at helping card manufacturers to clearly identify the assets to protect.
Thank you for your attention!
Questions?
During my PhD thesis, I have co-written **27 publications**:

- 2 book chapters;
- 5 journal articles;
- 3 invited conferences;
- 12 articles in international conferences with review and proceedings;
- 4 articles in national conferences with review and proceedings;
- 1 articles in national conferences with review and without proceeding;
- 1 posters.