Concolic execution on Java Cards

Using JPF and JDart as a concolic execution engine

August 2016

Marouan SAMI

INRIA – Team TAMIS

Table of contents

Introduction

On the path of expanding the connectivity of objects, and as one of the first steps toward concretizing the idea of IoT[1], Smart Card was born to enforce the rapid growth of wireless communication in a way to touch one of the most important and sensitive economic sectors by evolving a traditional in-store transaction business model to an on-line transaction model. However, the rapid emergence of those electronic ships and the big success that has known by the industry led the researchers to question about the safety, reliability and security of this device and a wide range of new applications build over it.

The main idea of this project consists of making Java Card testing as effective as possible and a less painful task. The use of concolic execution as a testing technique makes the system under test less obscure and more predictable, it also opens gates for more vectors of actions over regular unit testing methods, especially on hard to manipulate programs like Java Card Applets.

Java Card

A smart card is an embedded integrated circuit (Figure 1) composed of a processor, memory and I/O support. This fully qualified tiny computer can process and store information.
With an 8/16-bits CPU architecture, few Kbytes of ROM memory and some bytes of RAM memory, a smart card can ensure transactions and encrypted data exchanges.

1 IoT : Internet of Things

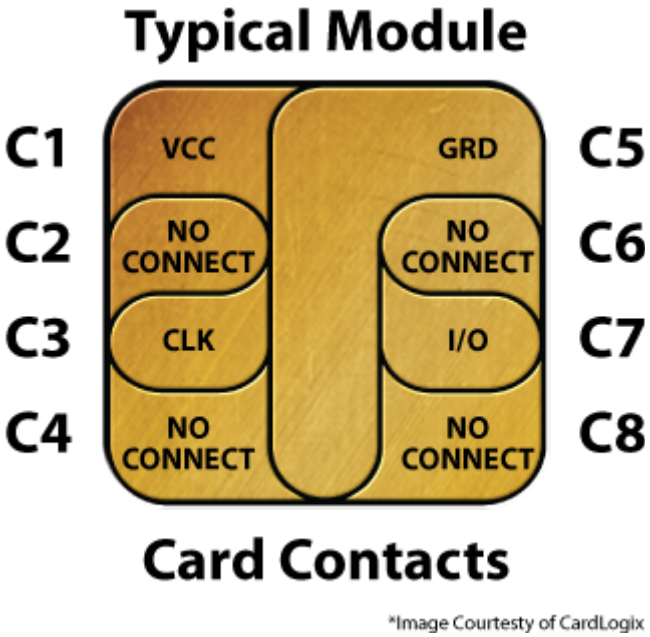


Figure 1 Smart Card contact pad

Java Card is a system that enables programs written in the Java programming language to be integrated into smart cards and other limited-resource devices.

With those limitations Java Card Technology had to design the Java System so that it conserves enough space for applications inside the smart card, that's why it supports only a subset of the features of the Java language.

The Java Card virtual machine is a split of two parts: off-card part and on-card part (Figure 2).

Many tasks such as class loading, bytecode verification, resolution and linking, are processed by the off-card virtual machine, where resources are not as limited as in the card.

Java Card Technology provides a runtime environment that manages the smart card memory, communication, security and application (Applet) execution. This runtime environment conforms to the smart card international standard **ISO 7816**.

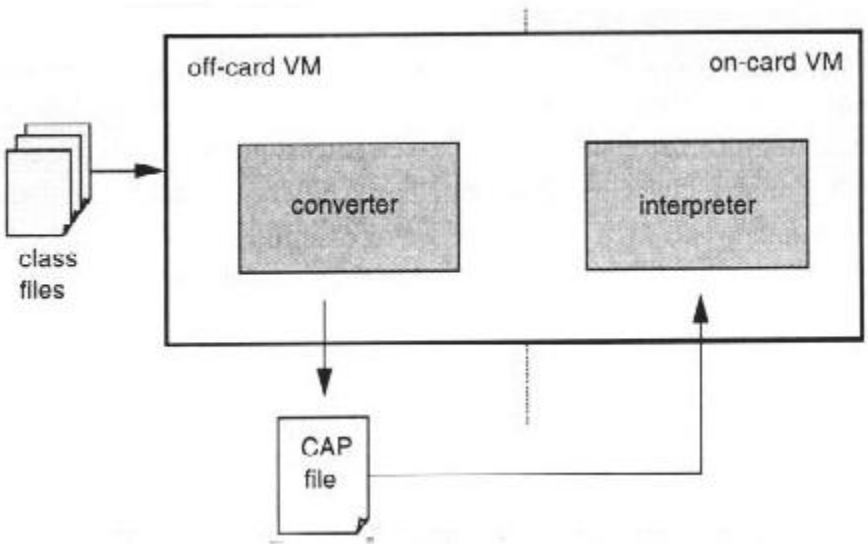


Figure 2 Java Card Architecture

Path exploration and Concolic execution

Java Path Finder: Path exploration

JPF is a state model checker for Java bytecode, which it means that JPF is basically a Java Virtual Machine that executes the program given in input as many times as necessary to explore every path it contains. Along the way JPF checks for property violations such as unhandled exceptions and deadlocks then it reports the trace that leads to it.

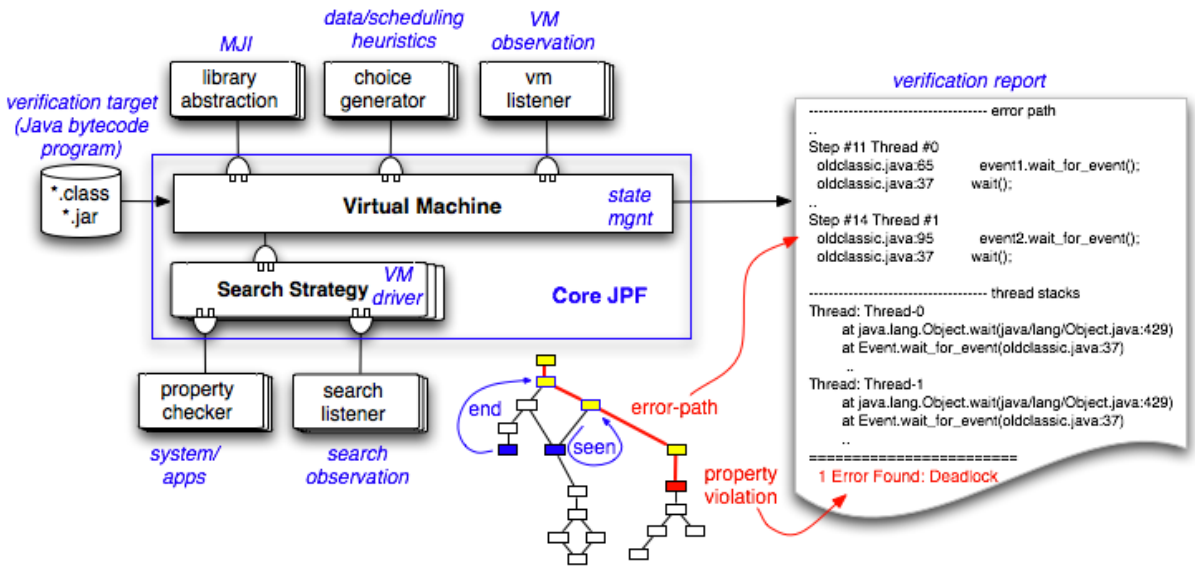


Figure 3 JPF model of operation

JPF does more than just tests a program for defects, it achieves model checking. However when trying to explore and execute all paths through a program, the number of those paths usually grows out of hand (which is known as *state explosion problem*), to resolve that, JPF do *state matching* which allows it to compare the current state at a choice point to an already seen state, in case the state is already checked then JPF is able to backtrack to a previous position that has unexplored paths and restore the program's state to that point.

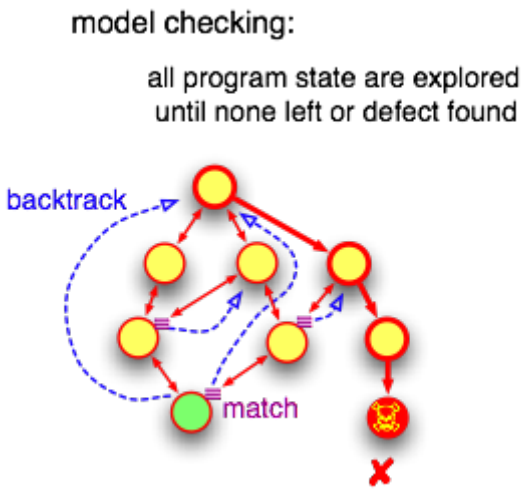


Figure 4 Path exploration

Model checking doesn't depend on guesses. In theory if there is a violation of a property then model checking will find it, since it is a method that exhaustively explores all possible behaviors, and that's from where it's effectiveness comes.

JDart: Concolic execution

Java Path Finder is not a black box tool. One of the greatest features of JPF is that it is fully customizable and extensible

framework. This aspect led to the creation of many useful extensions such as symbolic and concolic execution engines.

One of the best extension available for JPF is JDart[2], a concolic execution engine that executes Java programs using both concrete and symbolic values at the same time.
A concolic execution starts with a concrete input value. As the program execution proceeds, symbolic path constraints are formed for this particular execution. Those symbolic constraints are in fact logic formulae composed of a set of conditional statements that where collected along the way.
The next step is negating a sub-formula of the path constraint so we can build a new vector of concrete input values with the help of a constraint solver.
This new vector is used for another execution that will result in a new path definition.
This procedure can be repeated until all possible paths are consumed or a time limit is reached.

Concolic execution solves a big problem in concrete execution, it significantly reduces a search space in a way that search space explosion is no longer possible because the formulae generated by the symbolic execution will give a lead on what exact concrete value must be passed to satisfy a particular path, no need to blindly provide a huge number of inputs that may all of them execute the same path.
Concolic execution also has its effect on traditional symbolic execution, when a condition cannot be resolved symbolically

2 JDart : https://github.com/psycopaths/jdart

due to the complexity of the statement (complex mathematic equations, operations on float numbers or strings…) then it is replaced by a concrete value, and make it possible to proceed and go further in the path execution.

JDart is a tool build on top of JPF framework that brings the concolic execution feature, with its instruction factory it keeps track of symbolic expressions. The path constraint is then passed to an SMT[3] solver in order to get a satisfying valuation. JDart uses by default Microsoft z3 solver but also provides a solver abstraction layer in order to allow the integration of other constraint solvers such as Coral, dreal…

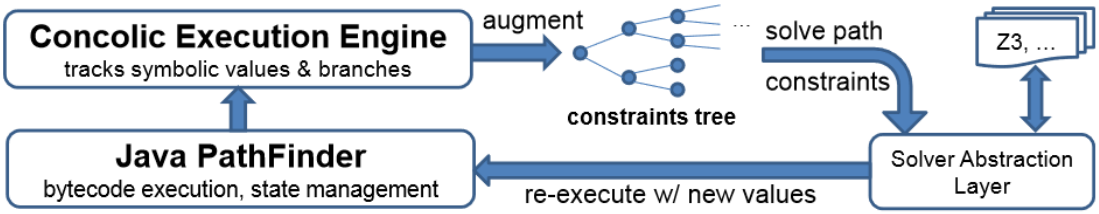


Figure 5 Architecture of JDart

Fig. 6 shows a Java method with two primitive parameters and two simple tests on these parameters, as well as the corresponding constraints tree produced by JDart. The constraints tree has one inner node for each condition in the code and three leaves. Since the method is of type void, ok-paths do not show post-conditions. The error-path is annotated with the class of the found error. We do not show the concrete inputs that were used to exercise these paths; JDart can be configured to display these values or generate unit-tests based on them.

3 SMT : Satisfiability Modulo Theories

```
public void foo(int i, boolean b) {
  if (i > 200000)
    if (b == false)
      assert false;
}

-('i' <= 200000)
  |-[+]_/OK: [  ]
  +-[-]-'b'
        |-[+]_/OK: [  ]
        +-[-]_/ERROR: java.lang.AssertionError
```

Figure 6 Example of JDart in action

The Java VM, the JPF VM and the JCVM

We defined JPF as a virtual machine that executes java programs and analyses its bytecode, so to be able to run the system under test, both Java virtual machine (referred to as host VM) and JPF VM cooperate and inter-communicate. The global architecture is quite complex, It's not always simple to know what java code get processed at which level. To make it worse, most of the standard libraries are processed by both VMs in different instantiations. Fig. 7 illustrate the different layers and what code is associated with the different layers.

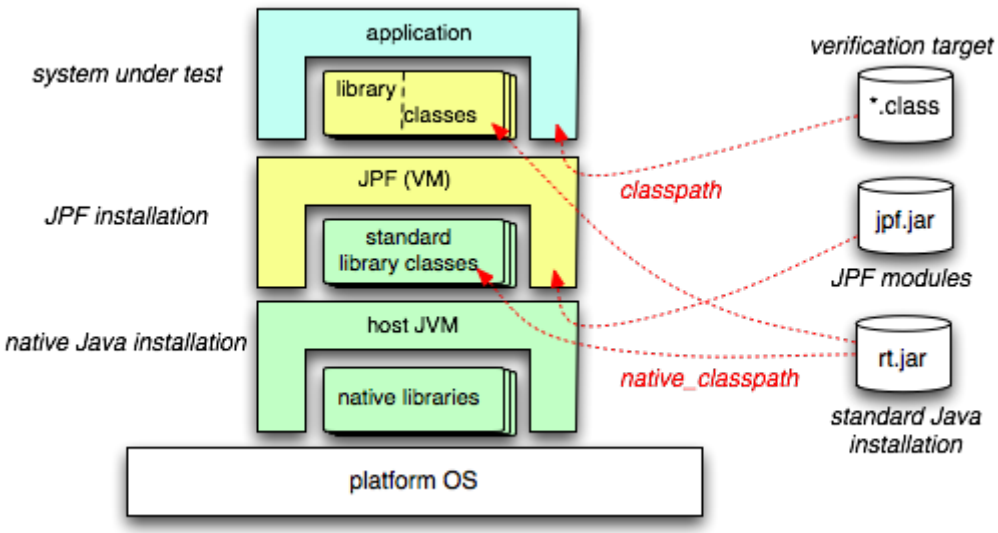


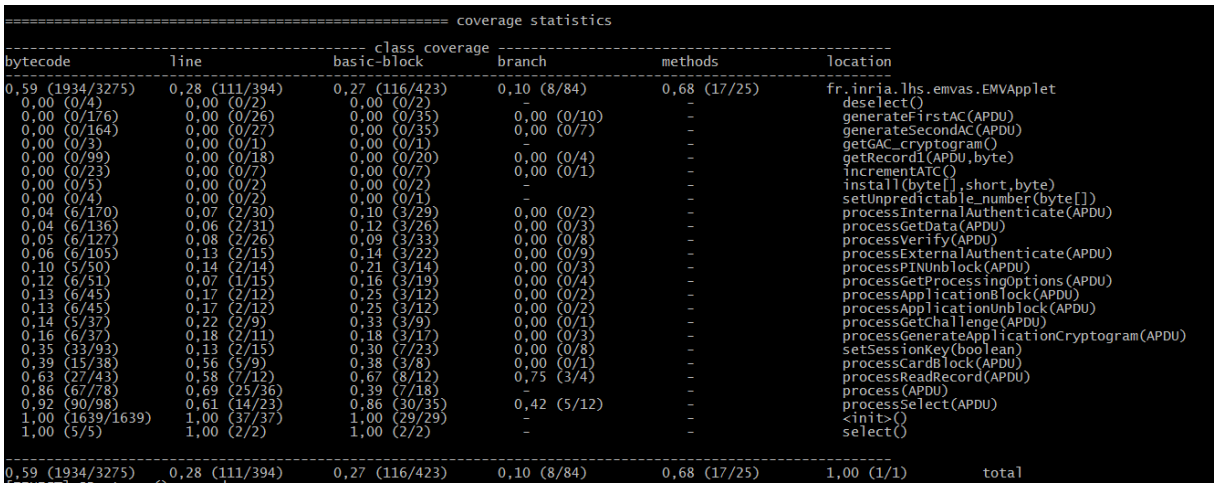
Figure 7 Different layers of the environment

On top of that combining the Java Card framework, and JPF and its extensions, requires a lot of background knowledge of the Virtual machine properties in both Java card and host JVM, Moreover to actually make tests using concolic execution on java card applets it is necessary to do some modifications on the Java Card API due to native methods that it contains.
Along the standard library classes, we have to add Java card libraries and integrate them inside the host JVM as well as inside JPF VM.

Testing a Java Card Applet

In this project we worked on several different applets, we tried to cover the widest possible range of features and functionalities of Java Card applets.
Running concolic tests on an applets requires to specify what fields should be treated concolicly and what should remain concrete.
Most of cases we want to treat the buffer of the APDU class as concolic since the requests that gets passed to the process method of the applet depends on the content of the buffer.
We focused particularly on the EMV standard, here are a resume of the results we got during our tests on a big EMV Applet designed to be the testing subject:

Code coverage

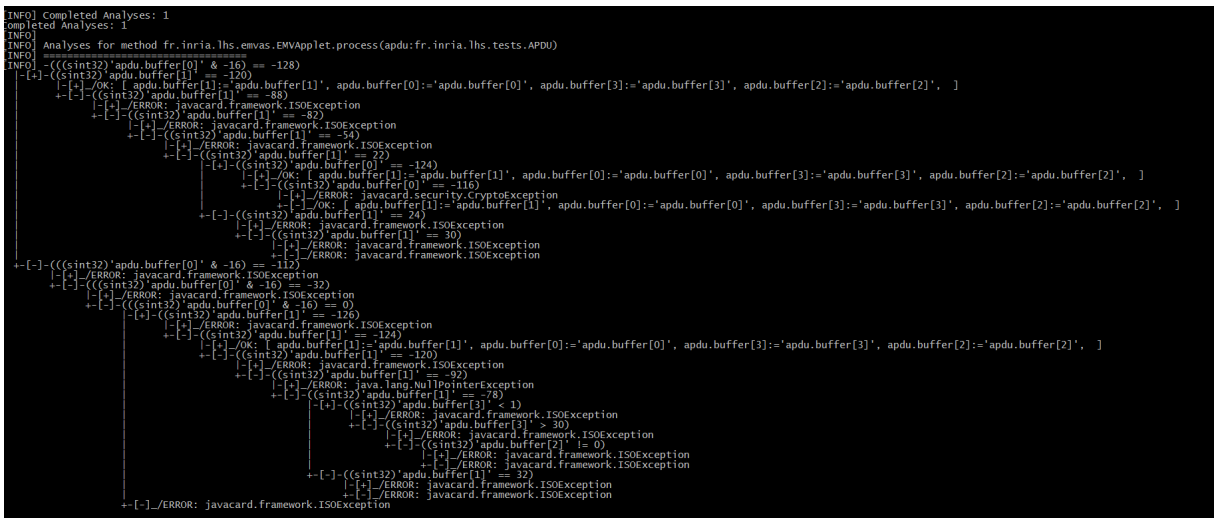


```
================================================ coverage statistics

------------------------------------------------ class coverage ------------------------------------------------
bytecode            line                basic-block         branch              methods             location
----------------------------------------------------------------------------------------------------------------
0,59 (1934/3275)    0,28 (111/394)      0,27 (116/423)      0,10 (8/84)         0,68 (17/25)        fr.inria.lhs.emvas.EMVApplet
  0,00 (0/4)          0,00 (0/2)          0,00 (0/2)          -                   -                   deselect()
  0,00 (0/176)        0,00 (0/26)         0,00 (0/35)         0,00 (0/10)         -                   generateFirstAC(APDU)
  0,00 (0/164)        0,00 (0/27)         0,00 (0/35)         0,00 (0/7)          -                   generateSecondAC(APDU)
  0,00 (0/3)          0,00 (0/1)          0,00 (0/1)          -                   -                   getGAC_cryptogram()
  0,00 (0/99)         0,00 (0/18)         0,00 (0/20)         0,00 (0/4)          -                   getRecord1(APDU,byte)
  0,00 (0/23)         0,00 (0/7)          0,00 (0/7)          0,00 (0/1)          -                   incrementATC()
  0,00 (0/5)          0,00 (0/2)          0,00 (0/2)          -                   -                   install(byte[],short,byte)
  0,00 (0/4)          0,00 (0/2)          0,00 (0/1)          -                   -                   setUnpredictable_number(byte[])
  0,04 (6/170)        0,07 (2/30)         0,10 (3/29)         0,00 (0/2)          -                   processInternalAuthenticate(APDU)
  0,04 (6/136)        0,06 (2/31)         0,12 (3/26)         0,00 (0/3)          -                   processGetData(APDU)
  0,05 (6/127)        0,08 (2/26)         0,09 (3/33)         0,00 (0/8)          -                   processVerify(APDU)
  0,06 (6/105)        0,13 (2/15)         0,14 (3/22)         0,00 (0/9)          -                   processExternalAuthenticate(APDU)
  0,10 (5/50)         0,14 (2/14)         0,21 (3/14)         0,00 (0/3)          -                   processPINUnblock(APDU)
  0,12 (6/51)         0,07 (1/15)         0,16 (3/19)         0,00 (0/4)          -                   processGetProcessingOptions(APDU)
  0,13 (6/45)         0,17 (2/12)         0,25 (3/12)         0,00 (0/2)          -                   processApplicationBlock(APDU)
  0,13 (6/45)         0,17 (2/12)         0,25 (3/12)         0,00 (0/2)          -                   processApplicationUnblock(APDU)
  0,14 (5/37)         0,22 (2/9)          0,33 (3/9)          0,00 (0/1)          -                   processGetChallenge(APDU)
  0,16 (6/37)         0,18 (2/11)         0,18 (3/17)         0,00 (0/3)          -                   processGenerateApplicationCryptogram(APDU)
  0,35 (33/93)        0,13 (2/15)         0,30 (7/23)         0,00 (0/8)          -                   setSessionKey(boolean)
  0,39 (15/38)        0,56 (5/9)          0,38 (3/8)          0,00 (0/1)          -                   processCardBlock(APDU)
  0,63 (27/43)        0,58 (7/12)         0,67 (8/12)         0,75 (3/4)          -                   processReadRecord(APDU)
  0,86 (67/78)        0,69 (25/36)        0,39 (7/18)         -                   -                   process(APDU)
  0,92 (90/98)        0,61 (14/23)        0,86 (30/35)        0,42 (5/12)         -                   processSelect(APDU)
  1,00 (1639/1639)    1,00 (37/37)        1,00 (29/29)        -                   -                   <init>()
  1,00 (5/5)          1,00 (2/2)          1,00 (2/2)          -                   -                   select()

----------------------------------------------------------------------------------------------------------------
0,59 (1934/3275)    0,28 (111/394)      0,27 (116/423)      0,10 (8/84)         0,68 (17/25)        1,00 (1/1)          total
```

Figure 8 Coverage statistics

We output some statistics related to the code coverage performed by the concolic execution, and we know that 59% of the bytecode get executed and that's a high rate.

Constraint tree



```
[INFO] Completed Analyses: 1
Completed Analyses: 1
[INFO]
[INFO] Analyses for method fr.inria.lhs.emvas.EMVApplet.process(apdu:fr.inria.lhs.tests.APDU)
[INFO] ================================
[INFO] -(((sint32)'apdu.buffer[0]' & -16) == -128)
  |-[+]-((sint32)'apdu.buffer[1]' == -120)
  |     |-[+]-/OK: [ apdu.buffer[1]:='apdu.buffer[1]', apdu.buffer[0]:='apdu.buffer[0]', apdu.buffer[3]:='apdu.buffer[3]', apdu.buffer[2]:='apdu.buffer[2]',  ]
  |     +-[-]-((sint32)'apdu.buffer[1]' == -88)
  |           |-[+]-/ERROR: javacard.framework.ISOException
  |           +-[-]-((sint32)'apdu.buffer[1]' == -82)
  |                 |-[+]-/ERROR: javacard.framework.ISOException
  |                 +-[-]-((sint32)'apdu.buffer[1]' == -54)
  |                       |-[+]-/ERROR: javacard.framework.ISOException
  |                       +-[-]-((sint32)'apdu.buffer[1]' == 22)
  |                             |-[+]-((sint32)'apdu.buffer[0]' == -124)
  |                             |     |-[+]-/OK: [ apdu.buffer[1]:='apdu.buffer[1]', apdu.buffer[0]:='apdu.buffer[0]', apdu.buffer[3]:='apdu.buffer[3]', apdu.buffer[2]:='apdu.buffer[2]',  ]
  |                             |     +-[-]-((sint32)'apdu.buffer[0]' == -116)
  |                             |           |-[+]-/ERROR: javacard.security.CryptoException
  |                             |           +-[-]-/OK: [ apdu.buffer[1]:='apdu.buffer[1]', apdu.buffer[0]:='apdu.buffer[0]', apdu.buffer[3]:='apdu.buffer[3]', apdu.buffer[2]:='apdu.buffer[2]',  ]
  |                             +-[-]-((sint32)'apdu.buffer[1]' == 24)
  |                                   |-[+]-/ERROR: javacard.framework.ISOException
  |                                   +-[-]-((sint32)'apdu.buffer[1]' == 30)
  |                                         |-[+]-/ERROR: javacard.framework.ISOException
  |                                         +-[-]-/ERROR: javacard.framework.ISOException
  +-[-]-(((sint32)'apdu.buffer[0]' & -16) == -112)
        |-[+]-/ERROR: javacard.framework.ISOException
        +-[-]-(((sint32)'apdu.buffer[0]' & -16) == -32)
              |-[+]-/ERROR: javacard.framework.ISOException
              +-[-]-(((sint32)'apdu.buffer[0]' & -16) == 0)
                    |-[+]-((sint32)'apdu.buffer[1]' == -126)
                    |     |-[+]-/ERROR: javacard.framework.ISOException
                    |     +-[-]-((sint32)'apdu.buffer[1]' == -124)
                    |           |-[+]-/OK: [ apdu.buffer[1]:='apdu.buffer[1]', apdu.buffer[0]:='apdu.buffer[0]', apdu.buffer[3]:='apdu.buffer[3]', apdu.buffer[2]:='apdu.buffer[2]',  ]
                    |           +-[-]-((sint32)'apdu.buffer[1]' == -120)
                    |                 |-[+]-/ERROR: javacard.framework.ISOException
                    |                 +-[-]-((sint32)'apdu.buffer[1]' == -92)
                    |                       |-[+]-/ERROR: java.lang.NullPointerException
                    |                       +-[-]-((sint32)'apdu.buffer[1]' == -78)
                    |                             |-[+]-((sint32)'apdu.buffer[3]' < 1)
                    |                             |     |-[+]-/ERROR: javacard.framework.ISOException
                    |                             |     +-[-]-((sint32)'apdu.buffer[3]' > 30)
                    |                             |           |-[+]-/ERROR: javacard.framework.ISOException
                    |                             |           +-[-]-((sint32)'apdu.buffer[2]' != 0)
                    |                             |                 |-[+]-/ERROR: javacard.framework.ISOException
                    |                             |                 +-[-]-/ERROR: javacard.framework.ISOException
                    |                             +-[-]-((sint32)'apdu.buffer[1]' == 32)
                    |                                   |-[+]-/ERROR: javacard.framework.ISOException
                    |                                   +-[-]-/ERROR: javacard.framework.ISOException
                    +-[-]-/ERROR: javacard.framework.ISOException
```

Figure 9 Constraint tree

We also get a graphical representation of all the paths and branches discovered during the process, this tree gives useful information on the conditions needed to be satisfied in order to follow a particular branch, we can also get the result of the tracked fields when we reach the leafs

Constraint statistics

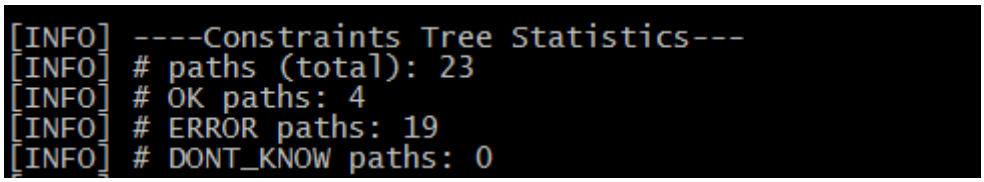


Figure 10 Constraint statistics

During the test we were informed that JDart actually executed 23 paths, 4 of them did not violate any property while the rest results in errors and unhandled exceptions.

Extending tests

Applying JPF to the SUT[4] gives a clear view of the structure of the applet tested, it also gives what are the exact conditions to be satisfied that causes exception raise and unexpected results. But what if we want to know if the applet follows the ISO 7816 specification and no forbidden transition from a state to another is violated?

One way to resolve this is to create a concolic fuzzer over JPF. This fuzzer would discover every path and every branch using the concrete values generated by the concolic execution.

Another alternative is to generate an automata of the SUT that describes all its states and transitions. Since JPF is a model checker concretizing this idea is relatively easy. This automata generated will be compared to the automata given by the ISO 7816 specification. Considering that the producer may generate his own specific states and transitions in his implementation of the specification. It is still possible to determine what is allowed from what is forbidden and what is safe from what may cause errors.

4 SUT : System Under Test

JFuzz & JDoop

In this section we will present 2 tools that we used while working on this project. The first one is JFuzz[5] an extension build by the NASA that do fuzzing using concolic execution on the SUT, it gets a seed concrete value as an input starts the concolic execution, calls a constraint solver and returns a concrete output with all the values generated.

Unfortunately, JFuzz is no longer effective as a tool, it's now obsolete and lost support since 2009. But its architecture is quite interesting and may give hints about how concolic execution can be designed.

The second tool is JDoop, this tool was designed as a mix of JDart and Randoop which is an automatic unit test generation framework.

So what JDoop adds is that Randoop no longer picks complete random input values for unit tests but instead, those inputs are feedback directed by the concolic execution done by JDart. This is good tool to test big projects like libraries or projects that contains a huge number of classes, but for a Java Card applet there is no added value.

Perspective

Another good extension offered by JPF community is JPF-nhandler[6] which delegates the execution of the native methods of the SUT to the host JVM and it catches back the result and send it to JPF in order to proceed with the execution.

---

5 JFuzz: http://people.csail.mit.edu/akiezun/jfuzz/

6 JPF-nhandler: https://bitbucket.org/nastaran/jpf-nhandler

With this extension, we could make the runtime environment of the Java Card applet as similar as possible to the real runtime environment inside the Smart Card. However to do this, we need to run the JCVM inside the computer that will run the tests, At the mean time it doesn’t seem to be possible.

Conclusion

Java Path Finder that is known as “the Swiss army knife of Java verification”, is indeed one of the best tools for testing a java applications, thanks to its extensiveness and ability to support and integrate new extensions. However the majority of its extensions are not designed to be used on java card system. This is an unexploited field that can make big contribution to the community.

Bibliography

Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.*

Dimjaševi, M., Giannakopoulou, D., Howar, F., Isberner, M., Rakamari, Z., & Raman, V. (2015). The Dart, the Psyco, and the Doop.

Documentation. (n.d.). Retrieved from Javapathfinder: http://javapathfinder.sourceforge.net/