Aalborg University

Master Thesis

Modelling Java Bytecode

Assessing Bit Flip Attacks and Countermeasures

Group:

Christoffer Ndũrũ

Kristian Mikkel Thomsen

Supervisors:

René Rydhof Hansen

Mads Christian Olesen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science

Software 10th semester

Address: Selma Lagerlöfs Vej 300

9220 Aalborg Øst

Phone no.: 99 40 99 40

Fax no.: 99 40 97 98

Homepage: http://www.cs.aau.dk

Project title:

Assessing Bit Flip Attacks and Countermeasures

Subject:

Modelling Java Bytecode

Project period:

1. Feb 2015 - 31. Jun 2016

Group name:

des107f16

Supervisors:

René Rydhof Hansen

Mads Christian Olesen

Group members:

Christoffer Ndũrũ

Kristian Mikkel Thomsen

Copies: 5

Pages: 64

Appendices: 3 & 0 CD

Finished: 31. Jun 2016

Abstract:

Attacks against smart cards are getting ever more advanced and it is thus important to stay ahead on the security front. Many different countermeasures exist to counter the attacks, but it can be difficult to choose appropriate countermeasures to protect a particular program. The purpose of this report is to address this concern by analysing how bit flips, induced by fault attacks, can affect Java programs. Previous work defining formal semantics and fault models for a small Java Card inspired language, are extended to provide a tool for automatic conversion of Java bytecode to UPPAAL models. We use UPPAAL to verify security properties with respect to formalised fault models.

Summary

In May 2011 the Economic Interest Group discovered that smart cards stolen in France were being used for transactions in Belgium. It turned out a group of criminals had managed to bypass the PIN verification on the cards and could use them for purchasing items, which they would later sell on the black market. Since smart cards are a widespread technology, for example in credit cards, abuse of them poses serious risks to both the banking industry, but also to consumers.

This report presents fault injections on the Java Card platform. It is based on previous work formalising a subset of Java Card bytecode and fault models. The architecture of the Java Card platform is presented and how persistent and transient faults can affect it.

An approach to automatic conversion of Java bytecode to UPPAAL models is also detailed. In extension, approaches for automatic modelling of a variety of fault injection attacks are described. Several known fault injection countermeasures are also presented, accompanied by a solution to automate model based safety analysis of Java Card programs, by inserting attacks into code modified with countermeasures, and modelling them in the modelling tool UPPAAL.

The solution uses the tool *Sawja* to provide a convenient representation of Java bytecode, which makes bytecode more readable. The tool is able to produce call graphs of programs, which can be used for automation of countermeasures. Improvements to the solution are also offered to allow future work, including suggestions for the first steps in automating implementation of control flow and control graph integrity countermeasures, which could be used to create a solution that can compare countermeasures' abilities to protect against fault injections.

A series of experiments are also conducted in an attempt to compare countermeasures' protection level against two selected code bases. In extension, we explore the viability of our experiment approach used to compare countermeasures.

Preface

This report is the result of a Master Thesis software engineering project at Aalborg University.

The aim of the report is to present a tool for converting Java bytecode into UPPAAL models, which in turn can be used to compare how vulnerable different programs are to fault injections, formalised by fault models. References are listed with numbers and not by the author's name(s), e.g. [23]. Furthermore, whenever writing "we", it refers to the project group and its members.

We would like to thank our supervisors for their guidance during the writing of this report.

Contents

CHAPTER 1

Introduction

In May 2011 Economic Interest Group noticed that smart cards stolen in France were being used in Belgium [5]. It turned out a group of criminals, aided by an engineer, managed to perform a man-in-the middle attack on the credit cards by placing a chip on top of the original chip. The attack bypassed PIN verification by intercepting communication between a credit card terminal and the original chip. As a consequence, a transaction would be approved whether the correct PIN was entered on the terminal or not. The gang is estimated to have caused damages below €600,000. They sold items purchased with the stolen cards on the black market, and managed to exploit over 7,000 transactions before being apprehended. As criminals are finding ever more sophisticated ways to exploit computer systems we rely on, on a daily basis, it is important to always stay ahead.

Smart cards are found in many places today, everywhere from phone SIM cards in phones, access cards and credit cards. The first wide spread use was French pay phone cards in 1983 [10, p. 366]. This report aims to investigate the security risks of fault injections aimed at Java Card, specifically attacks relying on random bit flips. We have built a tool for constructing UPPAAL [2] models from Java bytecode, which allows us to simulate random bit flips and countermeasures. The foundation of the model building is based on our previous work [4] with the TinyJCL language, found in Appendix B.1.

In this report we will assess three selected attack countermeasures: code duplication, call graph integrity and control flow integrity, which are expected to improve the security of Java Card bytecode.

CHAPTER 2

Faults

We consider two general categories of faults that can occur to a Java Card: *persistent* and *transient* faults. The main difference between these is that the persistent faults will affect the program every run, while the transient faults will only be present for a limited amount of time. Faults can occur when hardware is exposed to radiation sources, e.g. infrared light, laser, heat or cosmic radiation from space. Persistent faults in a piece of hardware, such as system, can occur in several ways, such as as a directed fault injection, e.g. a laser beam, targeting a persistent part of memory, such as the EEPROM of a Java Card. This could cause a bit flip in a value that is persistent across power-ups, and thus cause a persistent fault since the wrong value will always be used. If one wishes to create a persistent fault, precision is important, both to strike a persistent part of memory, but also to affect the correct value.

Transient faults do not cause permanent damage to the hardware. They can cause a temporary bit flip, resulting in a corrupted value, changed control flow causing unintended behaviour, or a crash of the hardware. The altered behaviour will disappear and the fault injection will have to be performed again, if the effect is to be reproduced. Nonetheless, both persistent and transient faults can have fatal consequences, if they strike at the right time and the right place, e.g. for an attacker trying to change a program's control flow and thus execute a sensitive piece of code.

The two categories of fault injection are thus sensitive to two variables, *time* and *place*, to different degrees. For example, an attacker who is trying to alter a constant in a program on a chipped access card, is able to work on the card in private surroundings. He can remove the protective layer on the chip and induce a persistent error on the card. Since the card is offline, the timing of the fault injection does not matter because he can only affect static values on the chip. The fault will still be present when the card is powered on at a later time.

On the other hand, an attacker who wants to change transient properties such as program flow dependent on a non-constant value, is very dependent on both timing and precision of his attack. He has to affect the correct place in memory at just the right time in the program's execution to alter the program flow. Table T2-1 illustrates the dependencies of persistent and transient fault injections.
It is important to note that when attacking the chip offline, the attacker only has access to persistently stored values. When attacking the card in online mode, the attacker also

	Persistent	Transient
Timing		X
Precision	X	X

Table T2-1: Table showing dependencies of induced faults.

has access to run-time related values, such as user input values stored on the operand stack.

It is also interesting to note that an attacker performing a fault injection on the chip while it is offline, has to leave the card in an uncorrupted state, since it will not boot up correctly if e.g. the *select* method, described in Section 2.1.1, of a Java Card was hit by the fault injection, thus corrupting it. This is another reason *precision* is important. When performing a fault injection on a card which is online, the attacker can strike both persistent values and run-time values. The attacker additionally does not have to leave the card in an uncorrupted state as he would have to when injecting a fault on an offline card. The reason is that the attacker might only need to flip a particular bit in e.g. a response APDU packet, see Section 2.1.1, or an operand stack value, to trick a card terminal into accepting a transaction. After the card has sent the manipulated packet, the attacker does not care whether the card crashes since the purpose of the attack has been served.

The time at which a bit flip is performed, matters in terms of how much of the system can be affected, e.g. at run-time versus when a card is powered off. Logically, a greater attack surface equals a greater probability of affecting a piece of memory that will cause a desirable outcome for an attacker. An example could be a method which is only runs once compared to a method which might be called ten times. Assuming methods of similar size and memory usage, the second method would have a probability ten times the that of the method which is only called once, to be hit by a bit flip. It should be noted that even if a method has a greater probability of being hit by a bit flip, it is not guaranteed that there is a greater chance that the bit flip will bring the card into a situation compromising security. This depends on the nature of the method.

2.1 Java Card

A Java Card is a smart card which are characterised by being a small embedded circuit which can process and store small amounts of information. The card does not have a power supply and does not work without a terminal to supply it with power. A well-known use of smart cards is to embed them in plastic cards and use them as e.g. credit cards. They have three types of memory: RAM, ROM and EEPROM. Information such as temporary variables can be stored in RAM and altered, but disappear after the chip is powered down. ROM memory cannot be altered and persists across power-ups

and downs. On a Java Card this is used to store the Java Card Virtual Machine. The EEPROM is persistent across power-ups and downs but information stored in it can be altered, and on a Java Card it is can be used to store third party applets and information used by the virtual machine.

The programs that run on a Java Card are called applets and a card can have multiple installed. They communicate with a terminal through Application Protocol Data Unit packets, described in Section 2.1.1. A credit card applet can for example inform a credit card terminal that an entered pin code is incorrect by sending it a packet, allowing a terminal to block a transaction.

2.1.1 Java Card Architecture

Applets installed on a Java Card run in the Java Card Virtual machine (JCVM) on the smart card. The hardware is limited and as a result, it is necessary that the JCVM is small in size. Most cards have 1.2kB of RAM, $32 - 48$kB ROM and 16kB EEPROM[9, Sec. 2.1]. To save resources, only a limited number of data types are supported, such as `short` and optionally integers while others such as `string` and `double` are not. If `string` and `double` were to be supported, code for performing string manipulation and floating point arithmetic would have to be included, which would take up valuable space.

The Java Card does not have its own power supply and applets must be protected from tears - an unexpected loss of power when a card is removed from a terminal. To handle this a transaction mechanism is provided, which allows a region of code to be atomically executed. If a tear occurs while an applet is executing in this region, operations performed by it are rolled back. This is useful in cases such as a credit card withdrawal process. If a card tear occurs after an internal balance counter is decremented, but before the withdrawal is registered, someone could potentially purchase items without actually paying. If the payment specific code region was protected by the transaction mechanism, the tear would have no effect.

The JCVM runs on top of the operating system, as illustrated in Figure F2-1, and supports a dialect of Java bytecode. On top of the JCVM, there is a final layer on top of which applets reside. The layer contains vendor and industry extensions, such as functionality used in the banking industry.

The JCVM offers features not found in Java virtual machines, such as a firewall separating the applets' memory. Since multiple applets reside side by side on the Java Card, it is vital to protect each applet's memory from other applets. If this was not the case, another applet could freely access and alter the memory of another applet, affecting its behaviour. If an applet needs an object from another, it can implement a shareable interface to expose selected objects. If the request is granted, the JCVM will perform a context switch, and the applet in which the object is residing will run the requested operation on the object. After the operation completes, the result is returned to the requesting applet.

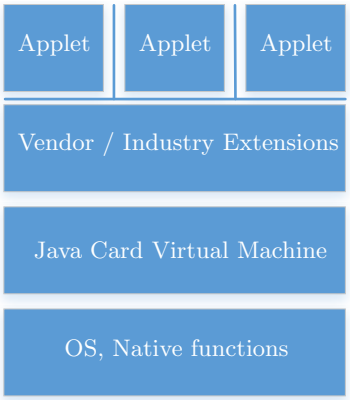


Figure F2-1: The Java Card architectural layers.

The Install Method

The `install` method creates an instance of the `Applet` subclass [9, API p. 65]. Depending on the application of the card, this method is called *once* in a card's life time, from either the manufacturer's or card distributor's side, after which applet installation on the card is disabled. Examples of such cards are SIM cards, since it could pose a security risk to allow other applets, than those intended to be on the card, to be installed.

The install method should perform all necessary initialisations and must perform a call to the `register` method. If the call is not performed successfully, or an exception is thrown before the call, the installation is not considered successful. If the installation fails, the Java Card runtime environment performs the necessary clean up actions when control is returned to it. After a successful installation, the applet can be selected with the `select APDU` command.

The Select Method

The `select` method is used inform the applet that it has been selected [9, API p. 68]. If initialisation, e.g. instantiation of objects on the heap, is required before processing APDUs, it should be done in this method. `Select` returns *true* to indicate success and it is ready to process the APDU.

APDU Packets

Communication between a smart card and a terminal is done via packets called Application Protocol Data Units (APDU). There are two types of communication packets: command and response. The structure of the packets can be seen in Table T2-2 and Table T2-3, respectively.

The command APDU has four mandatory header bytes (CLA, INS, P1, P2), some optional command data. L_c encodes the length of the command data, and L_e encodes the maximum number of bytes expected in the response APDU. The response APDU

has optional response data followed by two mandatory status bytes (SW1, SW2) indicating the status of the command request sent to the card, e.g. success.

CLA	INS	P1 - P2	L_c	Command Data	L_e

Table T2-2: The structure of a command APDU.

Response Data	SW1 - SW2

Table T2-3: The structure of a response APDU.

Communication is always initiated by the terminal by sending a command APDU - the Java Card may or may not reply to the command with a response APDU. Figure F2-2 illustrates that a command APDU packet is sent from a terminal to it. The command packet is then processed and a response might be sent from the card to the terminal.

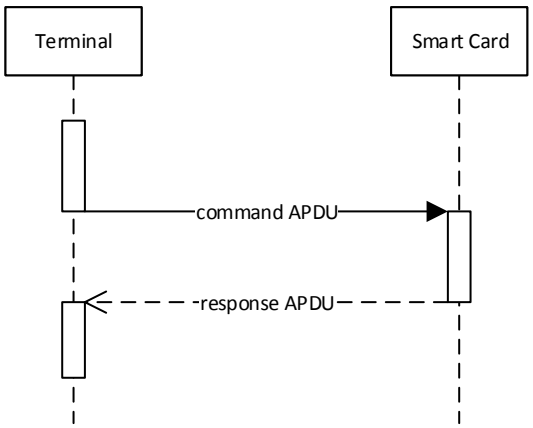


Figure F2-2: Communication between a terminal and a Java Card via command and response APDUs. Figure from [4, p. 4].

2.2 Countermeasures

Countermeasures can be implemented to protect applets against fault injections. Though they add security, they also increase the size of the applet source and degrade performance, resulting longer run time. In the following, common countermeasures are presented along with a consideration on how to increase effectiveness of the Field of Bit countermeasure.

2.2.1 Code Duplication

As described by [8] is a countermeasure which protects against corruption of data used for branching at execution time, such as local variables. It offers protection against

changes in a program's control flow, by duplicating instructions which are used to retrieve values, that affect branching.

An example could be a program, as seen in Listing 2.1, which loads two values from local variables, program counter (pc) 1-2, pushes them onto the operand stack and compares them. If the two values are the same, a jump is performed to a code region with sensitive code (pc 8-9), which approves a transaction on a credit card. Now, assume that a bit flip has occurred in a condition variable in pc 2. Before the flip, the rejection code would have been executed, but after the flip `acceptTransaction` is executed. When code duplication is implemented, as in Listing 2.2, redundant instructions are inserted. In pc 8-9, the values of the local variables are loaded again and pushed onto the operand stack. Afterwards, the comparison is performed again, and another jump is made to the sensitive code region in pc 15-16. It should be noted that this particular duplication only protects against a *single* bit flip in the original portion of the code. If a second bit flip occurs in the duplicated part of the code to affect the jump, the program can still execute `acceptTransaction`, even though it should not.

```
...
1: LOAD 1;
2: LOAD 2;
3: IF_CMPEQ 8;
<rejection code>
8: PUSH 0;
9: INVOKEVIRTUAL 12; // acceptTransaction();
...
```

Listing 2.1: Original program without code duplication implemented. The code is written in TinyJCL.

```
...
1: LOAD 1;
2: LOAD 2;
3: IF_CMPEQ 8;
<rejection code>
8: LOAD 1;
9: LOAD 2;
10: IF_CMPEQ 15;
<bit flip detected code>
15: PUSH 0;
16: INVOKEVIRTUAL 12; // acceptTransaction();
...
```

Listing 2.2: Modified program with code duplication implemented. The code is written in TinyJCL.

2.2.2 Call Graph Integrity

As described by [8], is a countermeasure which attempts to detect changes in the call graph of a program, e.g. caused by a bit flip. The idea is to have a unique id set by every caller before a call, which is checked by every callee to see if the call was made from a legal caller. This also works the other way around from callee to caller, where the the callee sets a unique id which is checked by the caller upon return.

In Listing 2.3, an example of the caller is shown. In pc 5 the unique caller id is pushed onto the stack and in pc 7 it is loaded into a class variable containing the id of the current caller. In pc 7 the method shown in Listing 2.4 is called. The caller id, 42, stored on the heap just before the method call, and the assigned id of the caller, already stored in another variable in memory are pushed onto the stack in pc 1 and 3 of 2.4. They are then compared in pc 2, and if the stored value is equal to the value set by the caller, the call graph integrity has been verified and the sensitive code beginning at pc 15 is executed. Note that these examples do not show the callee setting a unique id to be verified upon return to the caller. An example of this can be seen in Listing C.2

```
...
5: PUSH 42;
7: PUTSTATIC 2;
9: INVOKESPECIAL 42;
...
```

Listing 2.3: Caller with call graph integrity implemented. The code is written in TinyJCL.

```
1: GETSTATIC 2
3: PUSH 42;
5: IF_CMPEQ 15
8: <data corruption handling code>

// sensitive code region
15: ...
```

Listing 2.4: Callee with call graph integrity implemented. The code is written in TinyJCL.

2.2.3 Control Flow Integrity

As described by [1], is a technique which allows a programmer to annotate code to denote sensitive regions. The programmer then inserts flag variables which increment a counter, as shown in Listing 2.5. It is also possible to insert verification points where the counter is checked to verify the flag value, shown in Listing 2.6. The idea is that if a change in the call graph occurs, e.g. a method call is skipped because of a fault in the program counter, the flag will have the wrong value. A full implementation on the code sample can be seen in Listing C.4.

```
0.  GETSTATIC 3
3.  PUSH 1
4.  ADD
5.  PUTSTATIC 3
...
```

Listing 2.5: Java code example of the control flow integrity countermeasure incrementing the control flow flag.

```
...
 4.  LOAD 0
 5.  INVOKESPECIAL 32 // processVerifyPIN()
 8.  GETSTATIC 2
 9.  PUSH 5
13.  IF_CMPEQ 25
<bit flip detected code>
25.  RETURN
```

Listing 2.6: Java code example of the control flow integrity countermeasure checking the control flow flag.

2.2.4 Field of Bit

The Field of Bit countermeasure described by [11], detects changes in instructions from executable to readable, and vice versa. It uses custom annotations to denote sensitive code regions, which means that the JCVM must be modified for the countermeasure to work. It can detect two changes:

- An increase in the number of operands for an instruction
- A decrease in the number of operands for an instruction

Off-card, a Java class file is processed and a Field of Bit (FoB) is created. It contains information about program counter values and whether the data at that value are executable or readable. The FoB is saved as a custom component in the class file.

When a method is interpreted on-card, if a Field of Bit annotation is detected by the JCVM, it checks the method's byte array for inconsistencies with the FoB. For example, if an instruction is being executed, its parameters are checked in relation to the FoB at their respective locations. If one of the parameters were bit flipped from being readable to being executable, it would cause a discrepancy, and an error would be detected. In the same way, a bit flip causing an executable instruction to become readable would also be caught.

The weakness of this countermeasure is that it cannot detect if an instruction is replace by one with the same number of operands, e.g. `ifeq a` (one operand) replaced by `goto a` (one operand). This is called an indistinguishable replacement, but [11, p. 54] states that the probability of this in a Java Card application is 10%.

2.2.5 Instruction Differentiation

A single bit flip can change the instruction being executed, e.g. `ifeq` can be flipped to `ifne`, `iflt`, `ifgt` and `goto` in the Java Card bytecode instruction set. As mentioned in Field of Bit, this is particularly troublesome if the two instructions take the same number of parameters and put the same number of elements back on the operand stack. If this is the case, the operand stack will have the same number of elements on it regardless of whether the original or the altered instruction is executed, and the same number of parameters. It can therefore not be detected with e.g. a countermeasure such as Field of Bit. This can partly be remedied by modifying the underlying binary representations of the instructions in the Java Virtual Machine. This would work by changing the representations in such a way, that as few as possible instructions with the same number of parameters, and the same number of elements returned to the operand stack, are different by one bit. When a single bit flip changes an instruction, the chances that it is an undetectable change, in regards to required parameters and elements put back on the operand stack by it, are smaller than before modifying the representations.

2.3 Program Analysis with Sawja

We use the tool *Sawja* (Static Analysis Workshop for JAva) [6] to analyse Java class files, and to create a call graph. Information from the graph can then be used to provide information for rewriting purposes, e.g. the call graph integrity countermeasure, described in Section 2.2. Due to the Java Card version of *Sawja* not being public available, experiments are performed on Java code, but the principles remain the same.

Listing 2.7 shows Java code where two classes are present, `A` and `B`. `B` inherits `A`, and both implement the method `bar()`. The implementation of the method that will be called depends on the boolean value `b`. When the compiled class file of this code is processed for a call graph by *Sawja*, the result is as Listing 2.8 shows. The numbers that are listed just after the method calls, e.g. 40 are interesting, since they cluster method call targets, which can not always be resolved statically. In our case, we can use this to see which methods the call graph integrity countermeasure should be implemented in. If a method is not called, *Sawja* does not include it in the call graph. In the case with code where it is impossible to tell which method will be called at run-time as in Listing 2.7, *Sawja* includes both, as in line 3 and 4.

```
1 public void foo(boolean b){
2     (b ? new A() : new B()).bar();
3 }
```

Listing 2.7: Java sample.

```
1 void Sample.main(java.lang.String[]),22 -> void B.<init>()
2 void Sample.main(java.lang.String[]),12 -> void A.<init>()
3 void Sample.main(java.lang.String[]),40 -> short A.bar()
4 void Sample.main(java.lang.String[]),40 -> short B.bar()
```

Listing 2.8: Call graph generated by *Sawja*.

It is also able to pretty print the bytecode so that it becomes easier to read as in Listing 2.9. The tool also inlines the constant pool, as is evident in pc 7, 9, 12, 14 and 16. For example in pc 7, `new A` would normally be `new #id` where the id would be a method reference stored with identifier `id` in the constant pool. This inlining makes for a more compact representation of the bytecode. Additionally, dead code does not show up in *Sawja*'s output. Note that in the output, the target of a `goto` is expressed in terms of a relative offset as seen in pc 11, and not a absolute program counter as in regular Java bytecode. The instruction in line 11 will jump to the instruction at program counter 21.

```
public void foo ( bool 1 ) ;
     Concrete Method
     Not parsed

0.  iload 1
1.  ifeq 13
4.  new A
7.  dup
8.  invokespecial void A.<init> ( )
       A.<init>
11. goto 10
14. new B
17. dup
18. invokespecial void B.<init> ( )
       B.<init>
21. invokevirtual short A.bar ( )
       B.bar
       A.bar
24. pop
25. return
```

Listing 2.9: Sawja sample. Note that the numbers on the inner left side are program counter values.

CHAPTER 3

UPPAAL and Formal Verification

In this chapter, the semantics of TinyJCL and the fault models considered, are briefly described. The full language and fault model semantics can be seen in Appendix B, and are taken in full from from [4]. We have made minor contributions to the semantics, thus extending them. TinyJCL is a variation of the core Java Card bytecode language. By core, it is meant that most of the instructions in the full Java Card bytecode language can be built from the instructions in TinyJCL. The language was created to make it easier to model, because of the fewer instructions. The instructions in TinyJCL and fault models are briefly described in Section 3.1 and Section 3.1.3, respectively.

3.1 TinyJCL

TinyJCL is a small language based on Java Card bytecode. Just as Java Card, TinyJCL is a stack based language with a heap. TinyJCL was created and formalised in [4], and the semantics can be seen in Appendix B.1 It supports the majority of features of Java bytecode such as classes and method invocation, which are shown in Appendix A. A brief overview of the instructions can be seen in Section 3.1.

Program errors are not defined in TinyJCL, since it is assumed that if no legal rule is defined, the virtual machine will exit, e.g. if the current instruction is resolved to be `LOAD` i and the parameter i is not part of the domain for the *Local* function, representing local variables, e.g. there is no variable defined for i.

The original semantics did not capture the notion of private method calls and exceptions, TinyJCL is therefore expanded with rules for `INVOKESPECIAL` and `THROW`, to support invocation of private methods and exceptions.

Rule / Instruction	**Description**
NOP	No operation. Only increments the program counter.
PUSH *v*	Pushes parameter *v* on top of the stack.
POP	Removes and discards top element of the stack.
ADD	Adds the two top elements of the stack and pushes the result back onto the stack.
DUP	Duplicates the top element of the stack and pushes it onto the stack.
GOTO *a*	Jumps to a certain address in the program.
IF_CMPEQ *a*	Compares the two top stack elements and performs a conditional jump to *a*.
INVOKESTATIC *mid*	Calls a static method.
INVOKEVIRTUAL *mid*	Calls a virtual method.
RETURN	Returns from a method. If the stack is non-empty the top value is returned.
PUTSTATIC *fid*	Writes the top value of the stack to a class variable on the heap.
GETSTATIC *fid*	Pushes a class variable from the heap onto the stack.
LOAD *i*	Loads a local variable onto the stack.
STORE *i*	Stores a value from the stack in a local variable.
PUTFIELD *fid*	Stores a value from the top of the stack and stores it in a field in an object.
GETFIELD *fid*	Reads a field in an object and pushes it onto the stack.
NEW	Creates an object on the heap and pushes a reference to it onto the stack.

Table T3-1: Compact description of TinyJCL semantics. Note that stack refers to the operand stack.

3.1.1 INVOKESPECIAL

`INVOKESPECIAL` is used to invoke private methods and constructors. Before the invoke, the operand stack contains an object reference and arguments to pass to the invoked method. These are consumed by the invoke. The reference is then stored in the local variables followed by parameters. Additionally, a new stack frame with an empty operand stack and a program counter set to 0, is added to the call stack.

$$\text{INVOKESPECIAL}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{INVOKESPECIAL}\ mid' \qquad CP(mid') = pn \\ ops = (x_0, \ldots, x_n, objr, p_1, \ldots, p_{pn}) \qquad ops' = (x_0, \ldots, x_n) \\ loc' = [0 \mapsto objr, 1 \mapsto p_1, \ldots, pn \mapsto p_{pn}]) \end{array}}{\begin{array}{c} CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops', pc\rangle, \langle mid', loc', \epsilon, 0\rangle)\rangle \end{array}}$$

3.1.2 THROW

`THROW` describes when an exception is thrown. There are two cases handled, when the `catch` block is found in the same stack frame as the `throw`, and when it is found in another stack frame, e.g. its invoker. The `exceptionLookup` method handles these two cases: If the `catch` is found in the current stack frame, the program counter is set to the location of the exception handling code, the operand stack is cleared, the *objr* reference (for the exception object) is pushed back onto the stack and execution continues as per the Java Card Virtual Machine v2.2 specification [9, JcvmSpec p. 151]. If no appropriate handling block is found in the current stack frame, the frame is popped and the stack frame of its invoker is reinstated and the exception rethrown.

$$CS = (CS', \langle mid, loc, ops, pc\rangle) \qquad ops = (x_0, \ldots, x_n, objr)$$

$$exceptionLookup(P, CS) = \begin{cases} (CS', \langle mid, loc, (objr), pc'\rangle & if\ \ exceptDef(P, mid, pc, objr) = pc' \neq \bot \\ exceptionLookup(P, CS') & otherwise \end{cases}$$

$$\text{THROW}\frac{inst(P, mid, pc) = \texttt{THROW}}{CP, P \vdash \langle H, CS\rangle \Rightarrow \langle H, exceptionLookup(P, CS)\rangle}$$

3.1.3 Fault Models

We use fault models to formalise how fault injections affect our modelled execution. A number of fault models describing how various bit flips can affect the execution were also defined. A summary can be seen in Section 3.1.3 and the full definition can be seen in Appendix B.3. Each rule defines how a single bit flip will affect the program state. For multiple bit flips, a rule must be applied multiple times, but for the purpose of this report we will only focus on single bit flips.

Rule	Description
DATA_FAULT	Describes a change in the operand stack, local variables or the heap, caused by a bit flip.
PROGRAM_FLOW_FAULT	Describes a change in the program flow or method identifier due to a bit flip.
INSTRUCTION_FAULT	Describes a change from one instruction to another, caused by a bit flip.

Table T3-2: Fault semantics with a short description. Note that stack refers to the operand stack.

3.2 Property Verification

UPPAAL has its own query language used to verify properties of a model [2, p. 204]. The language is a simplified version of timed computation tree logic. UPPAAL's query language consists of *state formulae* and *path formulae*. The path formulae can be categorised into three categories: reachability, safety and liveness. These are described below, and they are summarised in Figure F3-1.

State formulae A state formula is an expression which can be evaluated for a state, without looking at the mode, e.g. $i \geq 42$. This formula asks whether it is true that i is greater than or equal to 42 in a given state. State formulae also allow one to verify whether a process is in a given location using an expression of the form `P.l`, where `P` is a process and `l` is a location in the process.

A deadlock is described using a special state formula, `deadlock`, and is satisfied for all states which deadlock.

Reachability properties express the notion that a state formula, φ, can *possibly* be satisfied on some path, going from the initial location of the model. In UPPAAL it is expressed as `E<>`φ. This could for example be used to verify whether a variable `i` in the model, along some path going from the initial location will have the value 2 by querying the model with `E<>i == 2`.

These types of properties are often verified as a part of a sanity check of a modelled system [2, p. 205], e.g. that it is possible to reach the done location in a Java Card program. Though this does not give any guarantee that the program will always finish, it makes sense to make sure to check whether it *possibly* can.

Safety properties state that "something bad will never happen". In other words, every state in a model will invariantly satisfy φ. This is useful e.g. to check that a bit flip *can* not cause a modelled program to end up in a location where e.g. incorrect credentials are authenticated and subsequently approved. Such an invariant safety property is expressed in UPPAAL as `A[]`φ, where the state formula, φ, would express

that the simulation of the model would never end up in the approved location when the credentials are incorrect.

A variant of this safety property, is one that expresses that "something will possibly never happen", e.g. a bit flip *might* not cause a modelled program to end up in a location where e.g. incorrect credentials are authenticated and approved. This is expressed in UPPAAL as `E[]`φ, which states that there should exist a maximal path[1], where φ is always true.

Liveness properties state that "something will eventually happen", e.g. verify that the program will eventually reach the end location. It is expressed in UPPAAL as `A<>`φ, and means that φ is eventually satisfied.

A variation of this liveness property, is the *leads to* property, written as $\varphi \rightsquigarrow \psi$. It is expressed in a UPPAAL query as φ `-->` ψ, and means that if φ is satisfied, ψ will eventually be satisfied, e.g. when Java Card transaction is begun, it will eventually end[2].

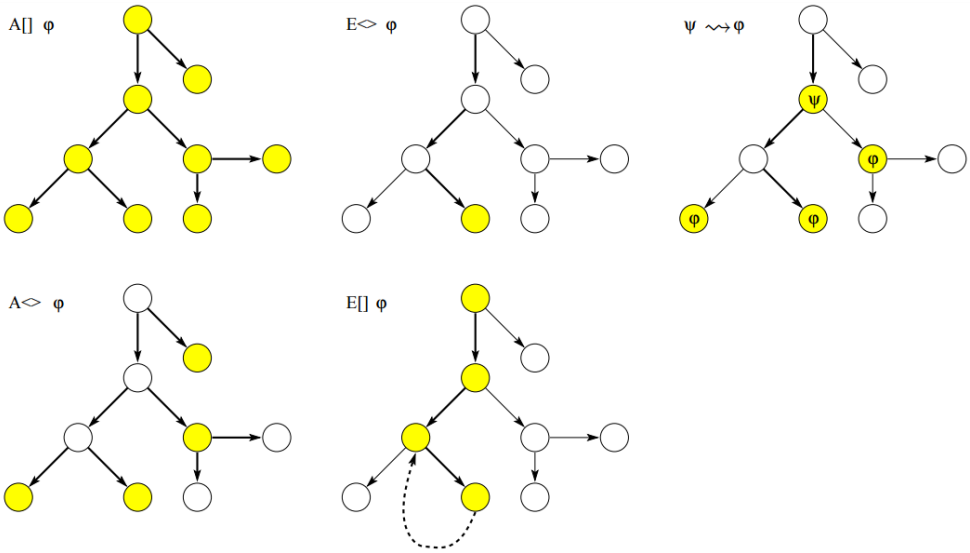


Figure F3-1: Illustration of the different property verification queries in UPPAAL. Taken from [2, p. 8].

Probability estimation UPPAAL SMC extends the capabilites of UPPAAL, in a way that allows us to reason about a model in terms of not only "yes" and "no", but also the *probability* a model has of entering a certain state. An example could be determining the probability a Java bytecode program model has of reaching a state of termination. To allow this, UPPAAL SMC extends regular queries, described earlier,

[1] A maximal path, is a path that is either infinite or the last state has no outgoing edges that can be traversed.

[2] A transaction in respect to Java Card, is a number of instructions which should be executed atomically.

to include probabilities. A probability query looks as follows:

`Pr[bound]` (φ)

where `bound` denotes a bound for a simulation run, within which a property, φ is to be verified. A bound can be defined in three ways [3, p. 402]

- implicitly by time by specifying $\leq M$, where M is a positive integer.
- explicitly by cost with $x \leq M$ where x is a specific clock.
- by number of discrete steps with $\# \leq M$.

UPPAAL SMC will then calculate the probability of this query being true within x runs and some confidence. The strength of probability queries is that in contrast to verification queries, they can easily be scaled depending on the desired precision and will give an answer, even on very large models.

3.3 Program Modelling

When translating a program to a UPPAAL model, several approaches are possible, depending on what is to be shown. One could for example represent a program merely in terms of program flow, if a disruption of program flow is to be simulated, e.g an error in the program counter. A memory corruption could be simulated by including the data flow in the model. These are just a few examples and many representations can be chosen. We have chosen the latter and model programs in terms of program and data flow to simulate disruptions in the execution flow.

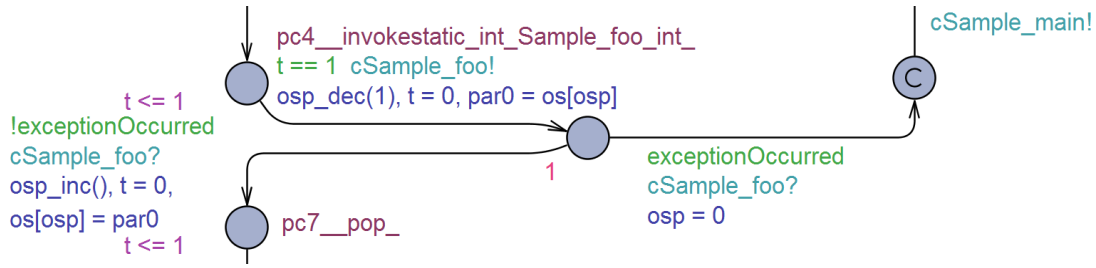
3.3.1 Program Conversion

The program conversion is based on TinyJCL semantics, most Java bytecode instructions can be translated directly to TinyJCL, but type information is lost. Because TinyJCL has operational semantics defined, it eases the implementation of the simulation, since each instruction is clearly defined.

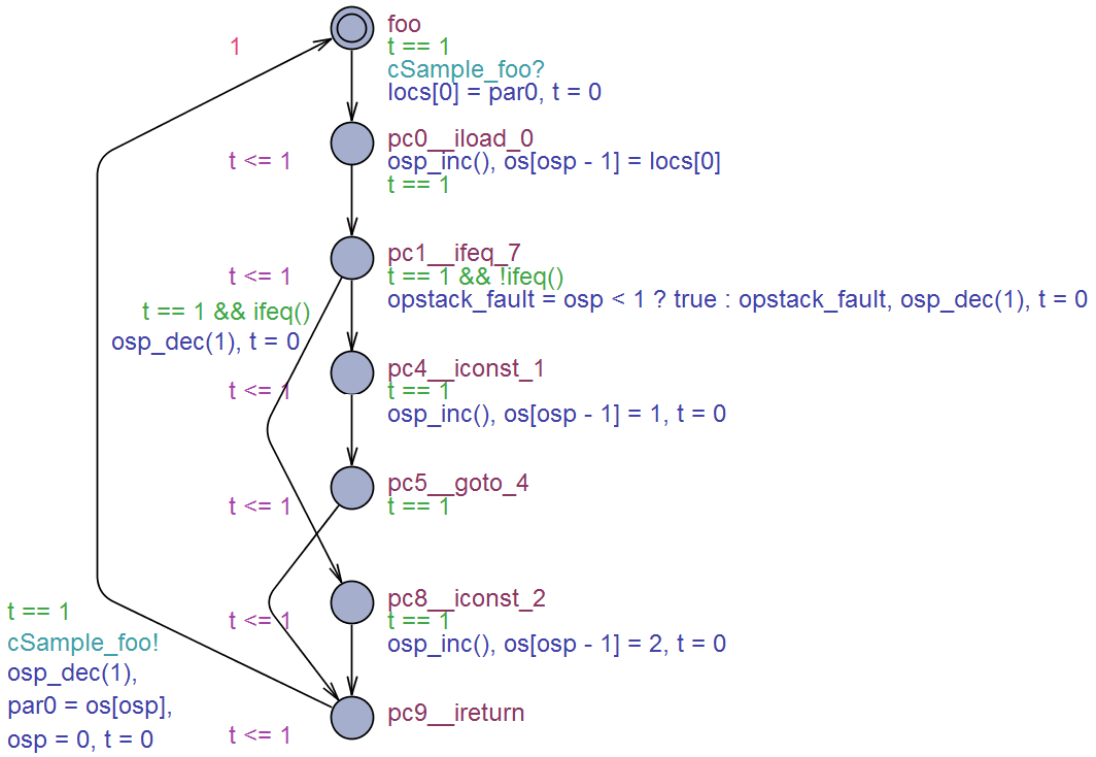
A Java method is represented by a template and method invocation is done by channels. This representation restricts the possibilities of recursion, and additionally the maximum memory usage must be known for allocation purposes. These limitations, while very restrictive for general Java, are already considered good practice for Java Card programming due the limited hardware, as discussed in Section 2.1. A program instruction, such as `ALOAD a` and `DUP` are represented as UPPAAL locations, this implies that a change in the program counter results in a change of location. In turn, this means that when an instruction is executed, the change to the program configuration *Conf* from Appendix B.1 occurs on the edge to the next location.

```
1  public class Sample{
2      public class Sample{
3          public static void main(String[] args) {
4              install();
5              foo(3);
6          }
8          public static int foo(int i){
9              return i != 0 ? 1 : 2;
10         }
11     }
12 }
```

Listing 3.1: Java code sample to be converted to a UPPAAL model.



(a) Method call in `main`.



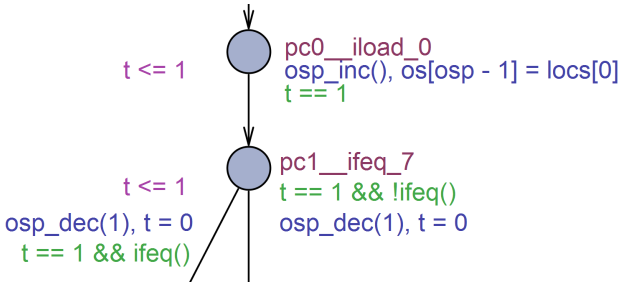
(b) the method `foo`.

Figure F3-2: Auto generated model of the `foo` method from Listing 3.1

Simple Instructions

```
0. iload 0
1. ifeq 7
...
```

(a) Java Bytecode Sample.



(b) Generated model from Sample.

Figure F3-3: Java bytecode and corresponding UPPAAL model.

Figure F3-3 shows how two Java bytecode instructions are represented in UPPAAL. On the left we see the Java bytecode. In the first line with pc 0, we have the `iload 0` instruction. `iload 0` pushes an integer from local variables at position zero on top of the operand stack, and increments the operand stack pointer and program counter. In UPPAAL the location `pc0_iload_0` represents `iload 0`. The UPPAAL model is seen in Figure F3-3b.

We have decided each non-library instruction to take 1 time unit, this is simulated with the location invariant `t <= 1` and guard `t == 1` on the edge leading to the next location. The guard is found below the location name, right of the edge and invariant is to the left of the edge. In this sample we defined the execution time as 1 time unit.

In the update on the edge seen below the guard, we simulate the data flow by assigning the value of the local variable, `locs[0]`, to the element at the top of the operand stack, `os`, represented by operand stack pointer, `osp`. `osp` is incremented as the operand stack grows and the increment of the program counter is simulated by the edge itself.

Jumps and Branches

For the majority of instructions, the program counter is set to the next instruction after execution. For a jump with `goto a`, however, the edge goes to the instruction with the program counter corresponding to the value of `a`, as seen in Figure F3-2.

Conditionals such as `if_cmpeq a` are modelled by a location having two outgoing edges, see Figure F3-3, one to the next instruction and one to the location associated with the current program counter plus offset *a*. On these edges, the guard is used to determine which of the edges is to be traversed.

The Operand Stack, Local Variables and the Heap

To simulate the operand stack, we use an operand stack pointer to point to the next free position in an array. Local variables and the heap are both represented as arrays in the

UPPAAL model, but they do not use an explicit pointer to access them since access to these are not necessarily performed in a top-down manner, as with the operand stack.

Method Calls

Method calls cover the following Java bytecode instructions: `invokestatic` for static calls, `invokespecial` for class constructors and private calls, and `invokevirtual` for virtual calls. To illustrate how these instructions are modelled, we use the Java code sample in Listing 3.2.

```
1  public class Virtual{
2    public Aclass a;
3    public Aclass b;
4
5    public Virtual(){
6      a = new Aclass();
7      b = new Bclass();
8      int ia = a.foo() + a.bar();
9      int ib = b.foo() + b.bar();
10   }
11 }
```

Listing 3.2: `Bclass` extends `Aclass`, `Aclass` implements the methods foo and bar, and `Bclass` overwrites foo.

The sample includes the bytecode instructions `invokespecial` and `invokevirtual`. `invokestatic` is omitted as `invokestatic` and `invokespecial` are nearly identical, the only difference is whether an object reference from the operand stack is passed as a parameter to the callee. As such, all method calls can be divided into two categories, virtual and static.

Static Methods

Static method calls, as shown in Figure F3-4a, are represented by three additions to the model. These additions consist of locations and edges.

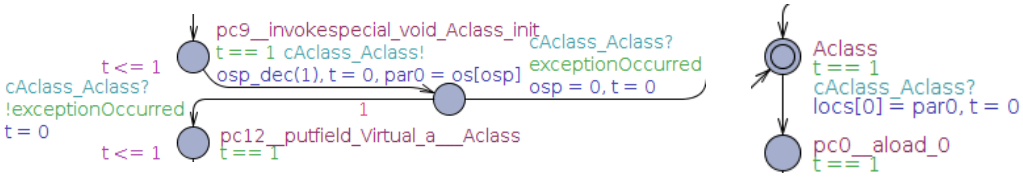
The first is a new location in the caller for every method call it performs. This makes it possible to simulate parameter passing from the caller, as well as control transfer when waiting for a callee to return control after a method call. The simulation of the caller remains in this location until the callee returns control, after its simulation has finished. This control transfer is modelled with a synchronisation on the edge, going from the new state in the caller and back to its original control flow, as seen in Figure F3-4b.

The second is an addition of one additional location in every template. The first, initial location, `Aclass`, in Figure F3-4c, serves two purposes: it enables the control transfer from the caller to itself by synchronisation, and simulates passing of arguments into the method from the caller.

The third is the edge from the `return` instruction, seen in Figure F3-4c. This is one of the two edges pointing to the `AClass` initial location, and the other is for exceptions, which are covered in Section 3.3.1. For main, the edge goes to a *Done* location instead of the initial location, where the simulation ends when it has finished. For other templates, this is where control is transferred back to the caller, and the edge goes to the initial location.

```
...
9. invokespecial void Aclass.<init> ( )
12. putfield Virtual.a : Aclass
...
```

(a) Invokespecial Bytecode generated by Sawja.



(b) Invokespecial Instruction.

(c) Initial Location.

Figure F3-4: Java bytecode and corresponding UPPAAL model.

Virtual Methods

Virtual methods are similar to static methods in regards to representation in the method template for caller and callee, but instead of handling control directly to callee method templates, a template responsible for resolving the virtual call is inserted for this purpose. Figure F3-5 is the resolver template generated for the code in Listing 3.2, there is a total of three virtual methods in this sample and the resolver has a waiting location for each.

Every class is mapped to an integer $clID$ and an array, $classHierarchy$, represents the class hierarchy of the program. The initial location `Invoke` waits for a synchronisation, after which the location `Resolver` has an outgoing edge for every possibility, in this sample that is five. There are essentially always three distinct possibilities

- There are no methods and no super where $clID == 0$.
- There is a method matching the class and method signature - in this case, call the method.
- There are no methods but there is a super class, then assign $clID = classHierarchy[clID]$ and try again.

This is based on *methodLookup* from the `INVOKEVIRTUAL` semantics in Appendix B.2.14. The usage of template for resolving method calls have similar limitation as representing

a template as a method, discussed in Section 3.3.1. It pose a problem when calling virtual methods from within a virtual method, but it can be handling by instantiating a instant of the lookup template for each virtual method, this however might scale poorly.

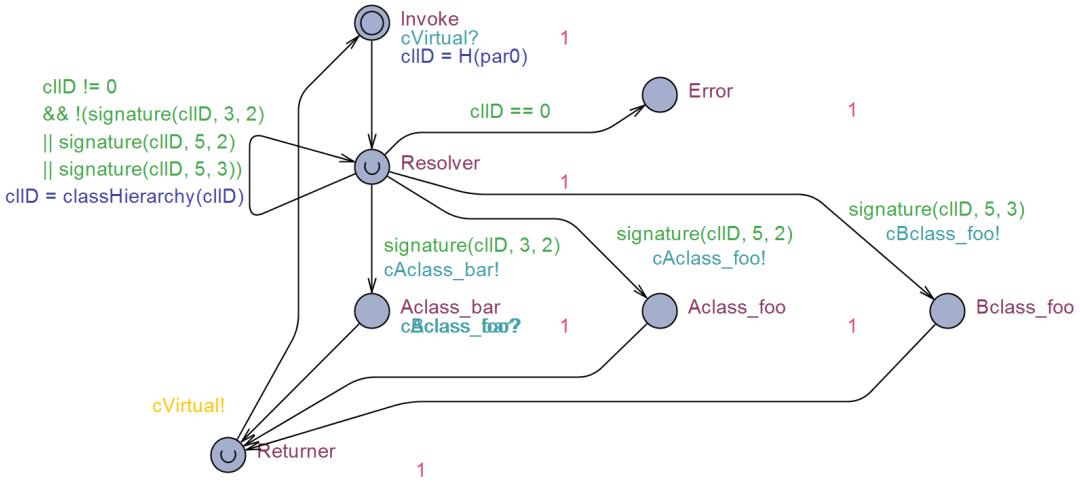


Figure F3-5: Invokevirtual.

Exceptions

Exceptions are handled by `try catch` blocks which do not have a corresponding bytecode instruction, instead they are defined at the end of a method as seen in Listing 3.3, `0, 8, 11` represent the corresponding program counters. When exceptions are thrown by the `athrow` instruction, there are two outcomes, either there is some exception handling covering the `athrow` defined in the current method, in which case the execution continues from the `catch start` pc. If there is no exception handling covering the `athrow` in the current method, the top stack frame will be popped and the process exception rethrown in its caller. On Java Card an unhandled exception thrown by an applet can be used to indicate an error such as *no access*.

The `athrow` location has an edge to a new location. The edge is required to set an *exception occurred* flag before passing control back to the caller by synchronisation. In Figure F3-2a the caller is shown, and if the *exception occurred* flag is set when assuming control, it will follow the edge to, in this example, the initial instruction. It is also possible for the `athrow` location to have an edge to a `catch` instruction, depending on whether an appropriate exception definition exists.

```
....
try start: 0; try end: 8: catch start: 11; catched type: java.lang.Exception.
```

Listing 3.3: Invokespecial Bytecode generated by Sawja.

3.3.2 Fault Modelling

We focus on three faults: program counter fault, data fault and static instruction fault where program counter and data fault are considered transient, and static instruction fault is considered persistent, as described in Chapter 2.

Program Counter Fault

To model a single bit flip occurring in the program's execution, a special fault template is introduced, illustrated in Figure F3-6. The template selects a random value between 0 and the maximum possible global clock value, which represents when in the program's execution a fault happens. The random value is assigned to a global variable in the UPPAAL system.

Every instruction in the Java bytecode is represented by a location, and has an associated program counter. There are edges from each location going to the locations which can be reached if one bit is flipped in the program counter. These edges have guards which check whether the time the fault is injected, corresponds to the global clock at the time the model simulation is at that particular edge. If it is, the guard will allow the edge to be traversed. There are no fault edges going back to the added locations described in Section 3.3.1, since these are not a part of the original program and therefore do not have an associated program counter.

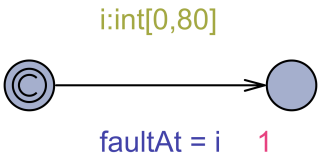


Figure F3-6: The UPPAAL template which selects when to perform a bit flip in the program counter

Data Faults

To model a data fault being injected into the operand stack, a special template selects when a fault should be injected into the operand stack. This happens in the same way as in the program counter fault described earlier, illustrated in Figure F3-6. The selected value is between 0 and the maximum possible runtime of the program. Figure F3-7 shows a method `getTriesRemaining`, in which a bit flip in the operand stack occurs. Edges going back to the locations `pc0_iconst_2` and `pc0_iconst_2` are where the faults are introduced. These are added by the solution and are not a part of an unaltered model of a program. The edges have a guard, $faultTime \leq globalClock$, determined by the special template, which only allows a fault to happen if the program execution has executed for a certain amount of time. The fault itself is introduced by the update $os[osPos]\hat{} = 1 \ll osBitPos$, which flips bit `osBitPos` of value at position `osPos` in the operand stack. `osBitPos` is a random value between 0 and 7, which denotes which bit should be flipp ed. `osPos` is a random value between 0 and the maximum size of the operand stack. After a fault has been introduced, the variable `faultTime` is set to

a value higher than the maximum value of the global clock, to ensure only one fault happens per simulation.

Our approaches to modelling faults in the operand stack, heap and local variables are similar and therefore only one of them is described.

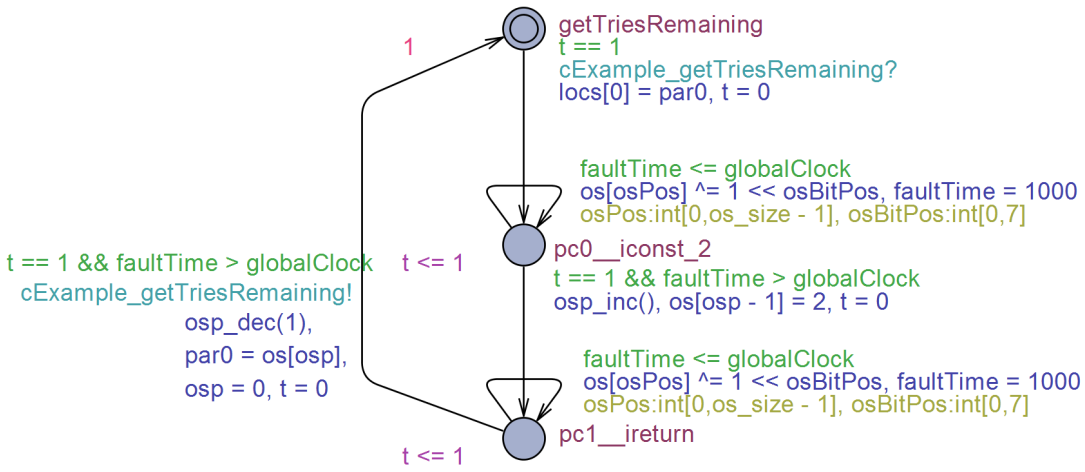


Figure F3-7: The UPPAAL model af a method where a bit flip occurs in the operand stack.

Instruction Fault

An instruction fault is not necessarily time-dependent, i.e. it does not rely on a clock as the *operand stack* fault described earlier. This fault model can represent both persistent and transient bit flips. The timing aspect of transient faults are modelled similar to operand stack faults, but persistent instruction faults are modelled by first assigning each edge of all templates a unique identifier. A special template then selects a random value in the range of 0 and the greatest identifier in the modelled program. A fault will only happen once since only one identifier is selected, and only when the simulation reaches the instruction chosen by the fault template. This is enforced using guards which compares the selected identifier with the current edge's. During the generation of the model, the solution has calculated all instructions, an instruction can be changed to by a bit flip in their binary encoding. Additional edges are then inserted to perform the actions of the altered instructions. For example, the `ifeq` instruction, which can be bit flipped to `goto`, would cause an extra edge representing the "yes" branch to be created to the destination location of the original "yes" branch. The new edge is different from the two original edges by always being enabled, thus causing a simulation to always perform a jump regardless of the result of the `ifeq` comparison.

3.4 Proposed Solution

We propose a solution which can convert a Java class file into a UPPAAL SMC model, modify the model to insert fault injection attacks and countermeasures, and assess the effectiveness of these. The purpose of the conversion is to be able to provide guarantees that the program has not become less secure with the implemented countermeasures.

The workflow stages of the solution are illustrated in Figure F3-8. The stages are labelled with numbers 1-3. Their purposes are detailed in the following.

Stage 1 - 2 parses Java bytecode through the solution's parser and generates a UPPAAL SMC model. This code can be either unmodified code or code implemented with countermeasures.

Stage 2 - 3 assesses the attacks performed and countermeasures implemented, to evaluate vulnerability and countermeasure effectiveness.

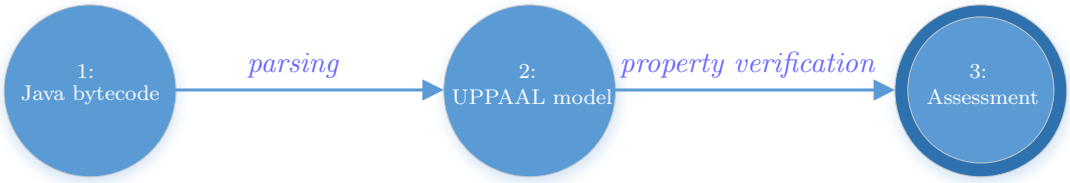


Figure F3-8: The workflow of the solution, from Java bytecode to UPPAAL SMC model and assessment of the modelled program.

CHAPTER 4

Experiments

In this chapter, experimental results are presented to investigate whether implementing code duplication, call graph and call flow integrity countermeasures, see Section 2.2, makes Java code more and not less secure. The experiments will be performed on code with and without countermeasures, implemented on a custom code sample and a mocked sample from the Java Card samples, found in Appendix C.

In order to determine whether the implemented countermeasures do indeed provide improved protection against bit flips, compared to unprotected code, experiments are needed. These are performed with the modelling tool UPPAAL, which is described in Chapter 3.

4.1 Java Card Purse

The experiments are performed on a code sample from a selected part of the Java Card samples seen in Listing C.1, where some parts, mainly variables and methods have been mocked in cases where it was not necessary or possible to model the complete sample.

The code sample is responsible for validating the PIN code, and in our experiments we will see how a bit flip can affect the validation. Experiments will be run according to three criteria: *No change*, *Crash* and *Attack*. We define *No change* as when the associated fault model never has any effect on the validation. *Crash* is defined as when the program reaches an illegal state, e.g. a misaligned operand stack or an illegal program counter. *Attack* is defined as when the validation is bypassed and the program remains in a legal state. Below, the types of queries and their purpose used are listed:

- No change: A **liveness** query checks whether the program can terminate in the correct state.
- Crash: A **reachability** query checks whether an *error* can occur.
- Attack: A **reachability** query checks whether the program can terminate successfully when an attack has occurred.
- Probabilities: For each state formula, a **probability** query is also run.

We assume that a bit flip will occur within 80 time units of program start. Through experiments, we discovered this number is greater than any of the individually modelled

programs. Additionally, the models are created in such a way that each instruction execution takes 1 time unit. This ensures a bitflip to occur within either one of the program's simulation runs, while the probability of affecting single instruction in any of the modelled programs are the same. The bit flip may or may not have an effect, depending on whether it occurs between program start and program end, or outside.

Results

The results in the table are listed in the following format in the extreme left column: $Code\ version + (attack)$, where attack is optional.

Version	No change	No change %	Crash	Crash %	Attack	Attack %
Base	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
Base + PC	False	[0.640, 0.650]	True	[0.342, 0.352]	True	[0.004, 0.014]
Base + OP	False	[0.990, 1]	False	[4,4e-5, 0.01]	True	[0, 0.01]
Base + H	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
Base + L	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CD	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CD + PC	False	[0.653, 0.663]	True	[0.331, 0.341]	True	[0.003, 0.013]
CD + OP	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CGI	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CGI + PC	False	[0.277, 0.287]	True	[0.701, 0.711]	True	[0.012, 0.022]
CGI + OP	False	[0.990, 1]	False	[0, 0.01]	True	[0.01, 0.01]
CFI	True	[0.990, 1]	False	[0, 0.01]	False	[0,0.01]
CFI + PC	False	[0.504, 0.514]	True	[0.478, 0.488]	True	[0.005, 0.015]
CFI + OP	False	[0.990, 1]	False	[0, 0.01]	True	[0,0.01]
CFI2	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CFI2 + PC	False	[0.558, 0.568]	True	[0.428, 0.438]	True	[0.004, 0.014]
CFI2 + OP	False	[0.990, 1]	False	[0, 0.01]	True	[2.266e-5, 0.01]

Table T4-1: Experiment results for Java Card Purse example. All experiment results have a confidence of 0.995

We do not utilise values from the heap or local variables to determine program flow, and as a result heap and local faults have no potential for attack. They are therefore omitted from the following experiment conclusions and results table, except in the Base version of the code sample. The base case for each countermeasure is provided to show that the countermeasure does not change the program's external behaviour.

Base

In the Base case, where no fault was introduced, the progam ran as expected.

In the case of Base + PC, there is a change in the crash and attack probabilities. The changes are likely due to two cases: When the operand stack becomes unaligned, e.g. because the program counter is changed to an address containing an instruction which consumes two stack elements, but the original instruction only consumes one. Or when a bit flip changes the program counter to an invalid value.

The results for Base + OP show that an attack is possible, however the probability simulation is not able to distinguish it from the Base case. This could be due to the fact, that the simulation did not enconter a run where an attack was possible.

Code Duplication

For CD + PC we expected a change in the attack probability compared to Base + PC, since the critical region of the code was offset, which might or might not have enabled new paths to be taken. The non-significant change in the results may be because the protected program happens to have the same amount of valid paths.

CD + OP successfully protects the code when it is subjected to a bit flip in the operand stack, as seen in the attack column. We attribute this to the fact that code that uses the operand stack, e.g. an `ifeq` instruction, is duplicated and therefore a flip in a value used, is overwritten with a correct value.

Call Graph Integrity

CGI + PC shows a higher vulnerability with a fault in the program counter, compared to the code duplication countermeasure. This is likely because the sensitive code region has become larger, as a result of additional instructions inserted to implement the countermeasure, thus resulting in a larger attack surface.

As CGI + OP shows, a successful attack is possible, but it can not differentiated from Base + OP, as the probability for a successful attack is very small.

Call Flow Integrity

Similar to CGI + PC, CFI + PC also has an increased crash probability, but without an increased attack probability compared to Base. It was expected that CFI might reduce the probability of a pc fault attack, as it introduces checks to confirm that important code has been run. We suspect the reason for this, is the structure of the code as the flag is only checked once in the experiment, and bypassing this check is enough for a run to be considered a successful attack.

CFI + OP shows no differences compared to base + OP. As expected, control flow integrity does not protect against errors in the operand stack.

Call Flow Integrity 2

CFI2 is a modification of the CFI example seen in Listing C.4, which was our attempt to reduce the attack probability, by reducing the number of instructions in the program to minimise the attack surface.

As the results show in the case of CFI2 + PC, the modification is hard to separate from the attack probability of CFI + PC, and it had no discernible effect. A positive side effect, however, was a reduced crash probability, which is likely due to the fact that the largest method in CFI2 has fewer instructions than in CFI, and thus a smaller attack surface.

CFI2 + OP is similar to CFI + OP, but a very small amount of attacks were successful in the simulation runs. This amount is neglible, however.

4.2 Invoke

The experiments are performed on the code sample from Listing 3.2, results can be seen in Section 4.2. The code duplication countermeasure are omitted because the example has no branches. The purpose of the sample is to include virtual and special invokes, to be used in these experiments. Because of the invokes, heap and locals faults now have an impact, and are therefore included in the result table.

Results

Version	No change	No change %	Crash	Crash %	Attack	Attack %
Base	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
Base + PC	False	[0.385, 0.395]	True	[0.607, 0.617]	True	[0.023, 0.032]
Base + OP	False	[0.961, 0.971]	True	[0.005, 0.015]	True	[0.013, 0.023]
Base + H	False	[0.912, 0.922]	True	[0.053, 0.063]	True	[0.004, 0.014]
Base + L	False	[0.966, 0.976]	True	[0.014, 0.024]	True	[0.004, 0.014]
CGI	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CGI + PC	False	[0.032, 0.042]	True	[0.859, 0.869]	True	[0.041, 0.051]
CGI + OP	False	[0.928, 0.938]	True	[0.005, 0.015]	True	[0.046, 0.056]
CGI + H	False	[0.821, 0.831]	True	[0,111 0.121]	True	[0.024, 0.034]
CGI + L	False	[0.937, 0.947]	True	[0.027, 0.0037]	True	[0.012, 0.022]
CFI	True	[0.990, 1]	False	[0, 0.01]	False	[0, 0.01]
CFI + PC	False	[0.147, 0.157]	True	[0.851, 0.861]	True	[0.007, 0.017]
CFI + OP	False	[0.938, 0.948]	True	[0.030, 0.040]	True	[0.016, 0.026]
CFI + H	False	[0.829, 0.839]	True	[0.131, 0.141]	True	[0.008, 0.018]
CFI + L	False	[0.964, 0.974]	True	[0.017, 0.27]	True	[0.01, 0.011]

Table T4-2: Experiment results for invoke example. All experiment results have a confidence of 0.995

Base

In the base case, no fault was introduced and as a result, the program ran as expected.

In Base + PC, the crash probability appear high which is likely because of misaligned program counter values.

The probability of crashes in Base + OP are present because the sample uses the operand stack to a greater degree than the Java Card Purse results, thus causing a larger attack surface. Results for Base + H and Base + L show detectable crash and attack probabilities, because the sample uses the heap for objects and local variables for storing calculation results.

Call Graph Integrity

Generally, the CGI countermeasure did not work well on the code sample, and the probability of a successful attack appears to be greater than for the Base example. We attribute this to the fact that the CGI code size was twice as big as the Base, and as a consequence created a larger attack surface, while CFI added significantly less size overhead.

Call Flow Integrity

For CFI + PC, the attack percentage is lower than that of CGI + PC, which indicates CFI is more effective at protecting against program counter faults. As a consequence, the crash probability is higher since an exception is thrown every time CFI detects a discrepancy in the call graph. Additionally, the extra instructions added by the countermeasure, result in more possibilities for the program counter fault to strike, but also increased possibility that the attack will cause an unaligned program counter value.

CFI + OP protects better against program counter faults than operand stack heap and local faults, but neither differentiate themselves in a significant way.

4.2.1 Summary

It would seem that code duplication and control flow integrity protect the Java Card purse sample code better than call graph integrity. This is because none of the faults introduced alter the call graph itself, they only change the program flow from one path to an already existing path.

A fault model we did not include is instruction parameter faults, such as flipping a bit in the target address of a `goto` or method index in an invoke. This could cause a change in the call graph. Additionally, if the examples had used virtual methods, there would be a chance of calling a method based on the wrong class id, caused by a bit flip. Call graph integrity would catch both of these cases.

In general, the CFI and CFI2 provide the best protection compared to the static size overhead. The CGI countermeasure adds an overhead larger than both CD, CFI and CFI2. CGI seemed to have little to no effect on the two code samples, which was due to the fact that few of the faults modelled could affect a method call. Fault models where we expect CGI to have a greater effect are bit flips in method identifier, see PFF_M Appendix B.3.2, bit flips in bytecode instruction parameters and bit flips during the method invokation.

CD provided a reasonable amount of protection, and incurred little overhead compared to Base. Table T4-3 shows a comparison of the static bytecode size compared to Base.

Version	CD	CFI	CFI2	CGI
Purse	1.25	1.55	1.45	3.07
Invoke	-	1.48	-	1.98

Table T4-3: Relative bytecode size of each Base with implemented countermeasures, compared to Base. Base is 1.0.

The small sample size of experiments we have performed, indicate that one has to be careful about where countermeasures are implemented. Thoughtless implementation

of countermeasures might not improve security of the code, and in the worst case make the code less secure.

CHAPTER 5

Conclusion

In this report an overview of the Java Card platform as well as attacks and countermeasures, aimed at the platform is presented. The primary focus is on bit flips, and how they can affect static and running behaviour of programs.

Our previous work [4] argued that it is difficult to measure and compare the effectiveness of countermeasures. We have tried to offer a solution to this problem by providing an automated approach for converting Java bytecode to UPPAAL SMC models, which allows us to test properties of a model, in its query language. We present an approach to model all major features of the Java bytecode language in UPPAAL. Our solution has some limitations, however. For example, it does not allow recursive calls and flips in method identifiers, which could be interesting to investigate in regards to countermeasures, e.g. causing endless recursion. The logic in our model conversion is based on TinyJCL semantics, which we have expanded upon with additional rules.

We have based our conversion on the Java static analysis tool *Sawja*'s native Java bytecode representation. In extension, we have investigated its call graph generation functionality, which is useful for automatic implementation of control flow based countermeasures.

In our experiments, we used our tool to construct models based on two code samples, and four fault models. Three fault injection countermeasures were also modelled for each code sample, and UPPAAL queries were then used to assess various properties of the models which has given us a basis for comparison of the countermeasures. A proof of concept instruction fault model was also modelled in UPPAAL for a few instructions, such as `ifeq` and `goto`.

We have shown that it is possible to automatically generate UPPAAL models of Java programs, and use UPPAAL's query language to verify security properties of the programs. Additionally, we have extended the models to include fault models, to evaluate the programs' vulnerabilities to fault attacks.

5.1 Future Work

The fault models should be extended by adding new fault models, to enable better assessment of countermeasures, for example control flow based countermeasures, such as call graph integrity. In addition, an analysis could be made to determine each attack's probability of success, relative to the other's, to aid in countermeasure selection. This would allow us to compare the fault models in terms of attack difficulty, which could be used in the selection of an appropriate countermeasure. Similarly, bit flips could be excluded from methods only being run at the manufacturer or vendor, such as the `install` method.

Additionally, further exploration of *Sawja*'s static analysis capabilities could prove useful in assessing countermeasures. We have not created a fault model for bit flips in the constant pool since *Sawja*'s bytecode representation hides the constant pool implementation. Since the constant pool has potential for allowing attacks, it would make sense to look into this.

More experimentation could be done to investigate which UPPAAL representation is most appropriate. For example, a single template approach where an entire program is modelled in one template, would simplify the implementation of invoke and bit flips in the method identifier. The current modelling and their consequent experiments only take single bit flips into account, this could be extended to include multiple bit flips, as malicious attackers could utilise more than a single bit flip to attack an applet.

APPENDIX A

TinyJCL Sample Model

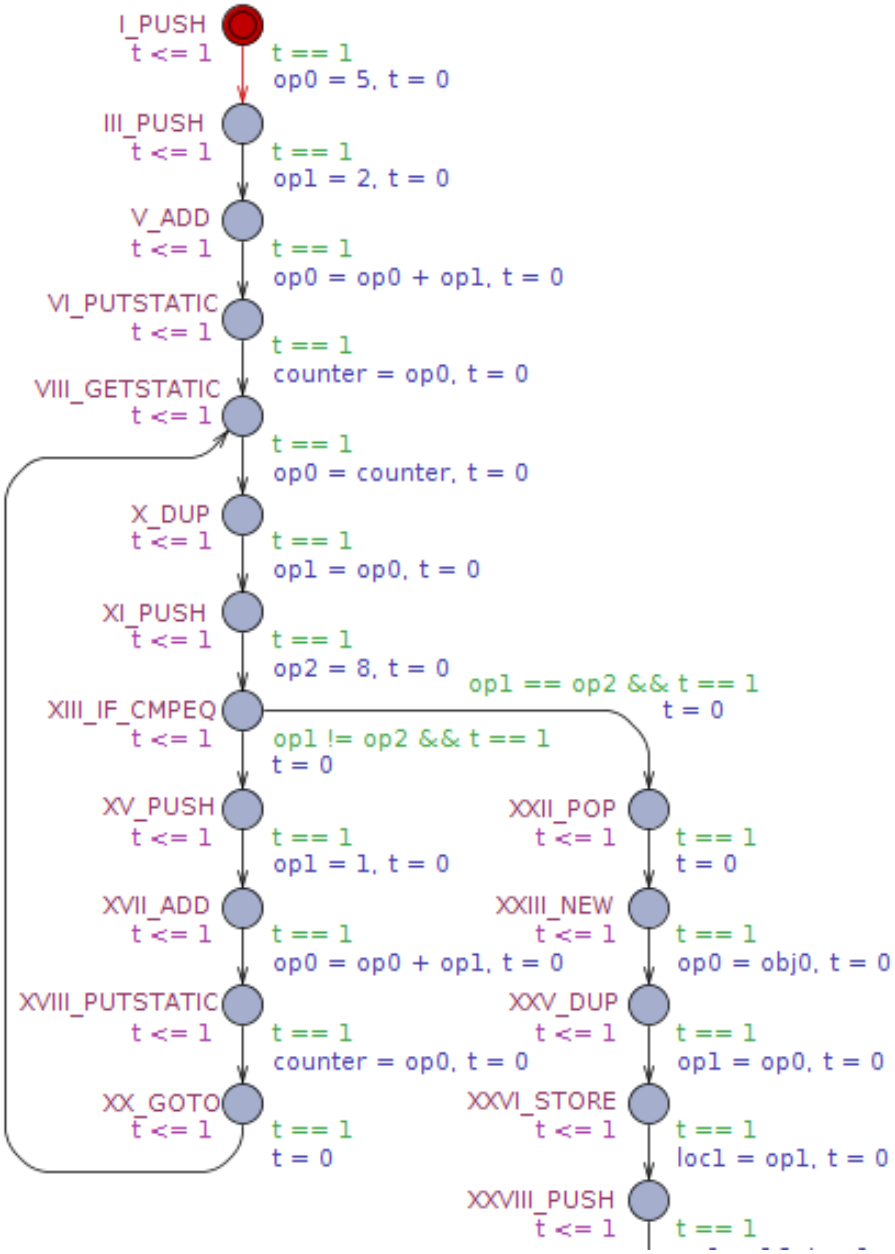


Figure A.1: Model of full TinyJCL

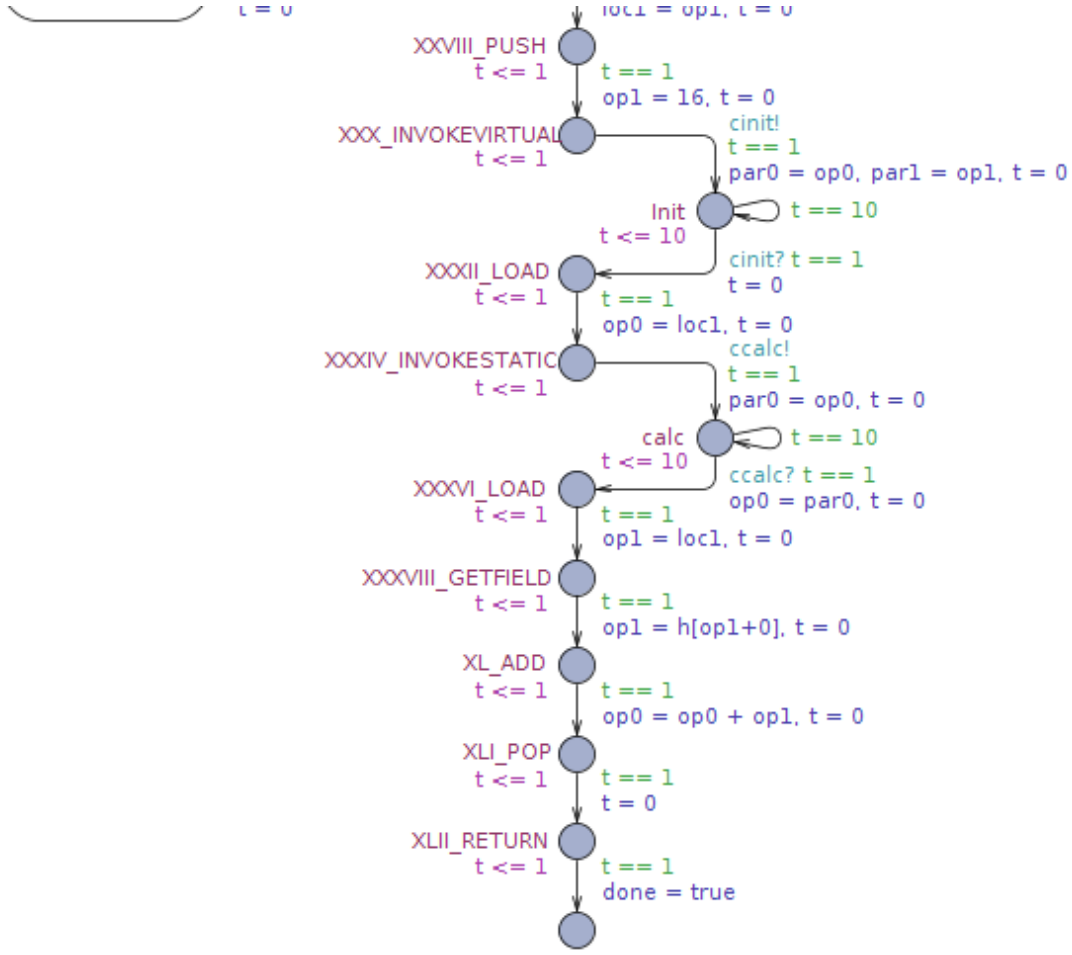


Figure A.2: Model of full TinyJCL

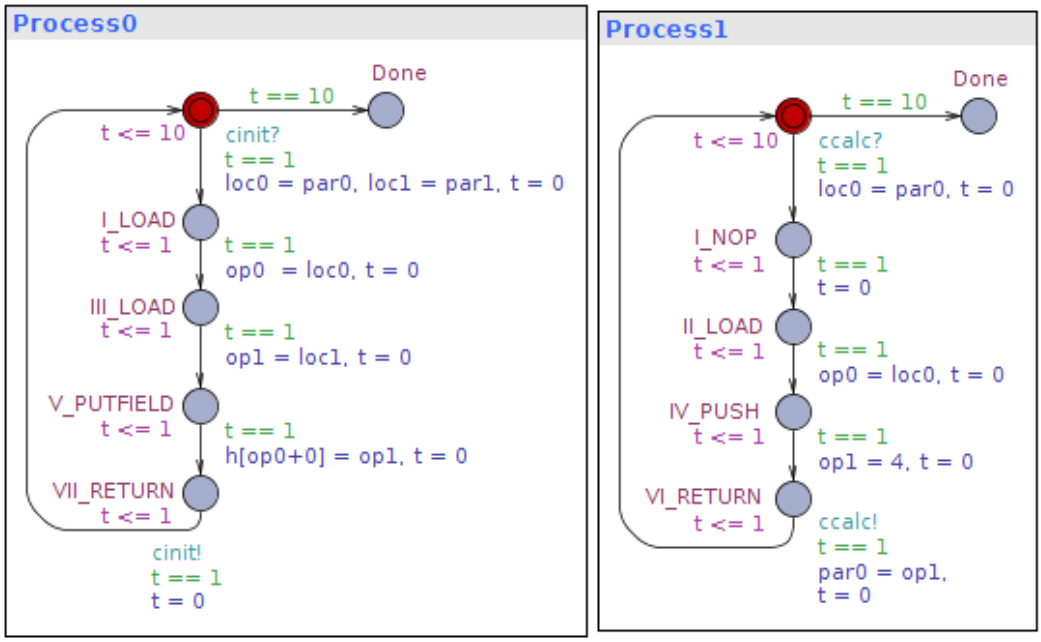


Figure A.3: Model of full TinyJCL

APPENDIX B

Semantics

Minor corrections have been made to the `RETURN`, `RETURN VOID` and `INVOKE_VIRTUAL` rules by the authors of this report.

B.1 Semantics

In this chapter we describe and formalise the language TinyJCL, which contains a variation of the core instructions of the Java Card bytecode language. In this context the term core describes the basic set of instructions from which all other Java Card instructions can be built. We created this language because it is easier to model the fewer instructions in this language, rather than all the instructions in Java Card. The full set of Java Card instructions can be built from combinations of the instructions in TinyJCL. Furthermore, there exists no official formal semantics for the Java Card language. The instructions of TinyJCL are defined as:

$$
\begin{aligned}
\mathit{Instructions} = \{&\texttt{NOP}, && \texttt{PUSH } v, && \texttt{POP},\\
&\texttt{ADD}, && \texttt{DUP}, && \texttt{GOTO } a,\\
&\texttt{IF\_CMPEQ } a, && \texttt{INVOKESTATIC } \mathit{mid}, && \texttt{RETURN},\\
&\texttt{PUTSTATIC } \mathit{fid}, && \texttt{GETSTATIC } \mathit{fid}, && \texttt{LOAD } a,\\
&\texttt{STORE } a, && \texttt{INVOKEVIRTUAL } \mathit{mid}, && \texttt{PUTFIELD } \mathit{fid},\\
&\texttt{GETFIELD } \mathit{fid}, && \texttt{NEW } \mathit{ci} && \}
\end{aligned}
$$

$\mathbb{N}$ is defined as the set of all natural numbers, including zero, and $\mathbb{Z}$ is defined as the set of all integers. In the operational semantics we want to describe values as an integer between a minimum value and a maximum value, $\mathit{Values} = \{x | x \in \mathbb{Z} \wedge x \geq \texttt{INT\_MIN} \wedge x \leq \texttt{INT\_MAX}\}$. In addition we want a notion of addresses which is used to refer to an instruction in a method and mapping to the heap: $\mathit{Addresses} = \mathbb{N}$. A program counter is used to represent the current address $\mathit{ProgramCounters} = PC = \mathit{Addresses}$. Instructions with parameters, such as `PUSH` v, increment the program counter with more than one, since it uses more than one byte.

The program is a sequence of instructions, we denote a program as $P = (i_0, \ldots, i_k)$ where k is the number of instructions in the program. A program consist solely of instructions $P \in \mathit{Programs}$ and $\mathit{Programs} = \{x | x \in \mathit{Instructions}^*\}$. To access instructions we introduce a function accepting a program, method identifier, and a

program counter. It returns the instruction in the method of the program at the program counter. The function is defined as:

$$inst = Programs \times MethodID \times PC \rightarrow Instructions$$

$$MethodID = \mathbb{N}$$

To describe a running program we use configurations. A configuration is a 4-tuple consisting of a program, constant pool, heap and a call stack.

$$Conf = Program \times ConstPool \times Heap \times CallStack$$

Executing an instruction means moving from one configuration to another. We will use $\vdash$ to indicate no change in the elements left of $\vdash$. For the semantic rules, no change will occur in program and constant pool e.g.:

$$CP, P \vdash \langle H, CS \rangle \rightarrow \langle H', CS' \rangle$$

Where $CP \in ConstPool$, $H, H' \in Heap$, and $CS, CS' \in CallStack$. We use a shorthand dot notation to access elements of a tuple e.g. $conf.Program$ where $conf \in Conf$, indicates the program used in the configuration $conf$.

The heap can be described as a function which takes a heap address and returns either the address *or* value associated with that address $Heap = Addresses \rightarrow (Addresses + Values)_\perp$. $\perp$ represents an undefined value, and is included to describe that $Addresses$ can also map to undefined addresses/values in the heap.

The call stack is used to keep track of the current method scope, it is a sequence of stack frames $CallStack = StackFrames^*$. A stack frame holds the method id, local variables, operand stack and the program counter for the method.

$$StackFrame = MethodID \times Locals \times OpStack \times PC$$

Local variables are represented by the function $Locals = \mathbb{N} \rightarrow Values_\perp$. The operand stack is a sequence of values and addresses $OpStack = (Values + Addresses)^*$.

To represent objects we need classes in our language. We represent classes as a 2-tuple with a possible super class and a function for resolving methods: $Class = Class_\perp \times Methods$. $Class_\perp$ is the super class or $\perp$ in the case of no super class $Methods$ is the set of all method identifiers implemented by the class. Object are represented by a 2-tuple with the class and fields of the object:

$$Object = Class \times Fields$$

Fields is a function for resolving the values of class variables:

$$Fields = FieldID \mapsto (Values + Addresses)$$

$$FieldID = \mathbb{N}$$

Finally we make use of a constant pool to resolve static method ids, fields of static classes and class definition when creating new objects:

$$CP = ConstPool = (MethodID \rightarrow \mathbb{N}) + (FieldID \rightarrow Addresses) + (ClassIndex \rightarrow Class)$$

$$ClassIndex = \mathbb{N}$$

B.2 Instruction Semantics

In the following semantics we make use of these abbreviation:

$$\begin{array}{rrr} H, H' \in Heap & CS \in CallStack & ops, ops', ops'' \in OpStack \\ mid, mid', mid'' \in MethodID & loc, loc' \in Locals & pc, pc' \in PC \\ v \in Values & a, objr \in Addresses & fid \in FieldID \\ obj, obj' \in Object & cl \in Class & \end{array}$$

B.2.1 NOP

The instruction has no other effect than incrementing the *pc*.

$$\text{NOP}\frac{inst(P, mid, pc) = \texttt{NOP}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc+1\rangle)\rangle}$$

B.2.2 PUSH

`PUSH` v is used to add the value from the parameter v onto the top of the operand stack. Since the `PUSH` v instruction takes up two bytes due to the parameter, *pc* is incremented by two.

$$inst(P, mid, pc) = \texttt{PUSH}\ v$$

$$\text{PUSH}\frac{ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+2\rangle)\rangle}$$

B.2.3 POP

This will remove and discard the top element of the operand stack.

$$inst(P, mid, pc) = \texttt{POP}$$

$$\text{POP}\frac{ops = (x_0, \ldots, x_{n-1}, x_n) \qquad ops' = (x_0, \ldots, x_{n-1})}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+1\rangle)\rangle}$$

B.2.4 ADD

This instruction consumes the two top elements of the operand stack, adding them together and pushes the result back onto the stack.

$$inst(P, mid, pc) = \texttt{ADD} \qquad v = x_{n-1} + x_n$$

$$\text{ADD}\frac{ops = (x_0, x_1, \ldots, x_{n-2}, x_{n-1}, x_n) \qquad ops' = (x_0 \ldots x_{n-2}, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+1\rangle)\rangle}$$

B.2.5 DUP

`DUP` duplicates the top element of the operand stack, leaving two identical elements as the two top elements of the operand stack.

$$inst(P, mid, pc) = \texttt{DUP}$$

$$\text{DUP}\frac{ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, x_n)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 1\rangle)\rangle}$$

B.2.6 GOTO

`GOTO` a takes an address as parameter and performs a jump to the specified address.

$$\text{GOTO}\frac{inst(P, mid, pc) = \texttt{GOTO } a \qquad pc' = a}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc'\rangle)\rangle}$$

B.2.7 IF_CMPEQ

This compares and consumes the two top elements of the operand stack. If they are equal it will make a jump to the address given as a parameter, otherwise it will increment pc by two.

$$inst(P, mid, pc) = \texttt{IF\_CMPEQ } a$$

$$pc' = \begin{cases} a, & \text{if } x_{n-1} = x_n \\ pc + 2, & \text{otherwise} \end{cases}$$

$$\text{IF\_CMPEQ}\frac{ops = (x_0, \ldots, x_{n-2}, x_{n-1}, x_n) \qquad ops' = (x_0, \ldots, x_{n-2})}{CP, P \vdash \langle H, (CS, \langle mid, locals, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, locals, ops', pc'\rangle)\rangle}$$

B.2.8 INVOKE_STATIC

`INVOKESTATIC` mid is used to call a static method. This involves adding a new stack frame on the call stack. The parameters of the methods are stored in local variables of that stack frame. These parameters are read from the operand stack. The number of parameters, pn, are found in the constant pool.

$$\text{INVOKESTATIC}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{INVOKESTATIC } mid' \qquad CP(mid') = pn \\ ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_{n-pn}) \\ loc' = [0 \mapsto x_{n-pn}, \ldots, pn \mapsto x_n]) \end{array}}{\begin{array}{l} CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops', pc\rangle, \langle mid', loc', \epsilon, 0\rangle)\rangle \end{array}}$$

B.2.9 RETURN

`RETURN` is used when returning from a method. The result of a `RETURN` depends on the state of the operand stack when called. If the operand stack is not empty the top element will be the return value.

$$\text{RETURN}\frac{\begin{array}{c} inst(P, mid', pc') = \texttt{RETURN} \qquad ops = (x_0, \ldots, x_n) \\ ops' \neq \epsilon \qquad ops' = (x'_0, \ldots, x'_n) \qquad ops'' = (x_0, \ldots, x_n, x'_n) \end{array}}{\begin{array}{c} CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle, \langle mid', loc', ops', pc'\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops'', pc + 3\rangle)\rangle \end{array}}$$

In following case, where the operand stack is empty, it will return without adding an element to the previous frame's operand stack.

$$\text{RETURN VOID}\frac{inst(P, mid', pc') = \texttt{RETURN} \qquad ops' = \epsilon}{\begin{array}{c} CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle, \langle mid', loc', ops', pc'\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops, pc + 3\rangle)\rangle \end{array}}$$

B.2.10 PUT_STATIC

This is used to write a value to a class variable on the heap.

$$\text{PUTSTATIC}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{PUTSTATIC}\ fid \\ CP(fid) = a \qquad H' = H[a \mapsto v] \\ ops = (x_0, \ldots, x_n, v) \qquad ops' = (x_0, \ldots, x_n) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.2.11 GETSTATIC

This reads the value of a class variable on the heap.

$$\text{GETSTATIC}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{GETSTATIC}\ fid \\ CP(fid) = a \qquad H(a) = v \qquad v \neq \bot \\ ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, v) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.2.12 LOAD

`LOAD` i is used to load the value of a local variable onto the operand stack.

$$\text{LOAD}\frac{inst(P, mid, pc) = \texttt{LOAD}\ i \qquad loc(i) = v \qquad v \neq \bot \qquad ops = (x_0 \ldots x_n) \qquad ops' = (x_0 \ldots x_n, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.2.13 STORE

This will store a new value in a local variable.

$$\text{STORE}\frac{inst(P, mid, pc) = \texttt{STORE}\ i \qquad loc' = loc[i \mapsto x_n] \qquad ops = (x_0, \ldots, x_{n-1}, x_n) \qquad ops' = (x_0, \ldots, x_{n-1})}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc', ops', pc + 2\rangle)\rangle}$$

B.2.14 INVOKE_VIRTUAL

`INVOKEVIRTUAL` is similar to `INVOKESTATIC` but in addition an object reference from the operand stack is stored as the first local variable, and the method for the actual class is resolved by a method lookup, inspired by [7]. For this we introduce two functions $signa$, and $methodLookup$. $signa = MethodID \rightarrow Signature$, where $Signature$ is the method's signature e.g. name and parameters. And $methodLookup$ used to lookup the intended method identifier, either from the class itself or a super class, defined as:

$$methodLookup(mid, cl) = \begin{cases} \bot & if\ cl = \bot \\ mid' & if\ mid' \in cl.Methods \wedge signa(mid') = signa(mid) \\ methodLookup(mid, cl.Class) & otherwise \end{cases}$$

$$\text{INVOKEVIRTUAL}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{INVOKEVIRTUAL}\ mid' \qquad CP(mid') = pn \\ ops = (x_0, \ldots, x_n, objr, p_1, \ldots, p_{pn}) \qquad ops' = (x_0, \ldots, x_n) \\ methodLookup(H(objr).Class, mid') = mid'' \qquad mid'' \neq \bot \\ loc' = [0 \mapsto objr, 1 \mapsto p_1, \ldots, pn \mapsto p_{pn}] \end{array}}{\begin{array}{l} CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \\ \qquad \langle H, (CS, \langle mid, loc, ops', pc\rangle, \langle mid'', loc', \epsilon, 0\rangle)\rangle \end{array}}$$

B.2.15 PUT_FIELD

`PUTFIELD` fid takes an object reference and a value from the top of the operand stack and stores the value in a specific field in the object.

$$\text{PUTFIELD}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{PUTFIELD}\ fid \qquad H(objr) = obj \\ H' = H[objr \mapsto obj'] \qquad obj' = obj.Fields[fid \mapsto v] \\ ops = (x_0, \ldots, x_n, objr, v) \qquad ops' = (x_0, \ldots, x_n) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.2.16 GET_FIELD

`GETFIELD` fid reads and consumes an object reference from the operand stack, and reads the value of the specified field in the object which is then stored on the operand stack.

$$\text{GETFIELD}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{GETFIELD}\ fid \\ obj = H(objr) \qquad v = obj.Fields(fid) \\ ops = (x_0, \ldots, x_n, objr) \qquad ops' = (x_0, \ldots, x_n, v) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.2.17 NEW

`NEW` ci creates a new object on the heap as well as pushing an object reference to the operand stack.

$$\text{NEW}\frac{\begin{array}{c} inst(P, mid, pc) = \texttt{NEW}\ ci \qquad CP(ci) = cl \\ obj = \langle cl, fields\rangle \qquad fields \in Fields \\ H(objr) = \perp \qquad H' = H[objr \mapsto obj] \\ ops = (x_0, \ldots, x_n) \qquad ops' = (x_o, \ldots, x_n, objr) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

B.3 Fault Semantics

We now introduce a fault model formalising the fault injections which can happen. This serves the purpose of showing exactly how a certain fault affects the Java Card,

whether it be a program flow change or a change in the memory.

To describe the fault semantics, we represent *Values* and *Addresses* as bit strings. We then use a Hamming distance of one, $\equiv_1$, to represent a bit flip, so $a \equiv_1 b$ means a differs from b with only one bit. It can also be stated as

$$\exists x \in \mathbb{N} : a \equiv_1 b \Rightarrow a = b \oplus 2^x$$

where $\oplus$ is the binary `XOR` operation. It can be expressed similarly with sequences

$$A \equiv_1 B \Rightarrow e \neq e' | A = (x_0, \ldots, x_{n-1}, e, x_{n+1} \ldots, x_m) \wedge B = (x_0, \ldots, x_{n-1}, e', x_{n+1} \ldots, x_m)$$

where n is the position of the differentiating element in the sequences A and B.

B.3.1 DATA_FAULT

A data fault can occur three places: the operand stack, the local variables or the heap. These faults are formalised in the `DF_OPS`, `DF_LOC` and `DF_HEAP` rules respectively.

$$\text{DF\_OPS}\frac{ops = (x_0, \ldots, x_n, \ldots, x_m) \qquad ops' = (x_0, \ldots, v, \ldots, x_m) \qquad v \equiv_1 x_n \qquad 0 \leq n \leq m}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc\rangle)\rangle}$$

$$\text{DF\_LOC}\frac{fid \in loc \qquad v = loc(fid) \qquad v' \equiv_1 v}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc[fid \mapsto v'], ops, pc\rangle)\rangle}$$

$$\text{DF\_HEAP}\frac{a \in H \qquad v = H(a) \qquad v' \equiv_1 v}{CP, P \vdash \langle H, CS\rangle \Rightarrow \langle H[a \mapsto v'], CS\rangle}$$

B.3.2 PROGRAM_FLOW_FAULT

This fault causes a change in the program flow of the applet. There are two cases: In the first case, either the fault changes the program counter, which has the consequence of changing which instruction is to be executed next. But since we have defined the program counter to only span locally within the method body, `PFF_PC` only describes a change in program flow within the method body. In the second case, `PFF_M` describes a fault which changes the method identifier, *mid*, of the method to be executed. The fault described by `PFF_M` will change the program flow to another method, outside of the current stack frame, but at the same program counter value within the new method's stack frame.

$$\text{PFF\_PC}\frac{pc' \equiv_1 pc}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc'\rangle)\rangle}$$

$$\text{PFF\_M}\frac{mid' \equiv_1 mid}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid', loc, ops, pc\rangle)\rangle}$$

B.3.3 INSTRUCTION_FAULT

This fault causes a change in an instruction, e.g. changing a `ADD` to a `POP` by changing a single bit in the instruction.

$$\text{INST\_F}\ \frac{\begin{array}{c} inst(P, mid, pc) = I \qquad P \equiv_1 P' \\ I, I' \in Instructions \qquad I' \neq I \qquad inst(P', mid, pc) = I' \end{array}}{\langle CP, P, H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle CP, P', H, (CS, \langle mid, loc, ops, pc\rangle)\rangle}$$

APPENDIX C

Code Samples

```
1  public class Example{
2      public static void main(String[] args){
3          try{
4          Example hw = new Example();
5          }catch (Exception ex){
7          }
8      }
10     public Example() throws Exception{
11         processVerifyPIN();
12     }
14     private void processVerifyPIN() throws Exception{
15         int pinLength = 4;
16         int faultCode = 255;
17         int triesRemaining;
19         short count = setIncomingAndReceive(); // get expected data
21         if(count < pinLength) throw new Exception();
23         if(isInvalid() != false){
24             triesRemaining = getTriesRemaining();
25             throw new Exception();
26         }
27     }
30     private boolean isInvalid(){
31         return true;
32     }
34     private short setIncomingAndReceive(){
35         return 5;
36     }
38     private int getTriesRemaining(){
```

```
39          return 2;
40      }
41  }
```

Listing C.1: Mocked Java example code from the Java Card samples

```
1  public class ExampleCGI{
2      private static int callId;
3
4      public static void main(String[] args){
5          try{
6              callId = 1;
7              ExampleCGI hw = new ExampleCGI();
8
9              if(!(callId == 2))
10             {
11                 throw new Exception();
12             }
13
14         }
15         catch (Exception ex){
16
17         }
18     }
19
20     public ExampleCGI() throws Exception{
21         if(callId != 1){
22             throw new Exception();
23         }
24
25         callId = 2;
26
27         processVerifyPIN();
28
29         if(callId != 3){
30             throw new Exception();
31         }
32
33         callId = 2;
34     }
35
36     private void processVerifyPIN() throws Exception{
37         if(callId != 2){
38             throw new Exception();
39         }
40
41         int pinLength = 4;
42         int faultCode = 255;
43         int triesRemaining;
44
45         callId = 3;
46
47         short count = setIncomingAndReceive(); // get expected data
48
49         if(callId != 4){
50             throw new Exception();
51         }
```

```
53          if(count < pinLength) throw new Exception();

55          callId = 4;

57          if(isInvalid() != false){
58              if(callId != 5){
59                  throw new Exception();
60              }

62              callId = 5;
63              triesRemaining = getTriesRemaining();

65              if(callId != 6){
66                  throw new Exception();
67              }

69              throw new Exception();
70          }

72          callId = 2;
73      }


76      private boolean isInvalid() throws Exception{
77          if(callId != 4){
78              throw new Exception();
79          }

81          callId = 5;

83          return true;
84      }

86      private short setIncomingAndReceive() throws Exception{
87          if(callId != 3){
88              throw new Exception();
89          }

91          callId = 4;
92          return 5;
93      }

95      private int getTriesRemaining() throws Exception{
96          if(callId != 5){
97              throw new Exception();
98          }

100         callId = 6;

102         return 2;
103     }
```

```
104 }
```

Listing C.2: Mocked Java example code from the Java Card samples with the call graph integrity countermeasure implemented

```
1  Class Example

3  private bool isInvalid ();
4  Concrete Method
5  Parsed

7  Example.processVerifyPIN()
8  0.  iconst 1
9  1.  ireturn

11  public Example (); Concrete Method  Parsed     Example.main(java.lang.String
       [])
12 0.  aload 0
13 1.  invokespecial void java.lang.Object.<init> ()
14 4.  aload 0
15 5.  invokespecial void Example.processVerifyPIN ()
16 8.  return

18  private void processVerifyPIN (); Concrete Method Parsed     Example.<init>()
19  0.  iconst 4
20  1.  istore 1
21  2.  sipush 255
22  5.  istore 2
23  6.  aload 0
24  7.  invokespecial short Example.setIncomingAndReceive ()
25 10.  istore 4
26 12.  iload 4
27 14.  iload 1
28 15.  ifcmpge 11
29 18.  new  java.lang.Exception
30 21.  dup
31 22.  invokespecial void java.lang.Exception.<init> ()
32 25.  athrow
33 26.  aload 0
34 27.  invokespecial bool Example.isInvalid ()
35 30.  ifeq 16
36 33.  aload 0
37 34.  invokespecial int Example.getTriesRemaining ()
38 37.  istore 3
39 38.  new  java.lang.Exception
40 41.  dup
41 42.  invokespecial void java.lang.Exception.<init> ()
42 45.  athrow
43 46.  aload 0
44 47.  invokespecial bool Example.isInvalid ()
```

```
50.  ifeq 4
53.  goto -20
54.  return

 public static void main ( java.lang.String [] 0); Concrete Method Parsed
 0.  new  Example
 3.  dup
 4.  invokespecial void Example.<init> ()
 7.  astore 1
 8.  goto 4
11.  astore 1
12.  return

try start: 0; try end: 8: catch start: 11; catched type: java.lang.Exception.

private int getTriesRemaining ();
Concrete Method
Parsed

Example.processVerifyPIN()
0.  iconst 2
1.  ireturn

private short setIncomingAndReceive (); Concrete Method Parsed    Example.
    processVerifyPIN()
0.  iconst 5
1.  ireturn
```

Listing C.3: Java bytecode example of the code duplication countermeasure

```
public class ExampleCFI{
    private static int flag = 0;

    public static void main(String[] args){
        try{
            ExampleCFI hw = new ExampleCFI();
        }catch (Exception ex){

        }
    }

    public ExampleCFI() throws Exception{
        processVerifyPIN();

        if(flag != 3){
            throw new Exception();
        }
    }

    private void processVerifyPIN() throws Exception{
```

```
21          flag++;

23          int pinLength = 4;
24          int faultCode = 255;
25          int triesRemaining;

27          short count = setIncomingAndReceive(); // get expected data

29          if(count < pinLength) throw new Exception();

31          if(isInvalid() != false){
32              triesRemaining = getTriesRemaining();
33              throw new Exception();
34          }
35      }


38      private boolean isInvalid(){
39          flag++;
40          return true;
41      }

43      private short setIncomingAndReceive(){
44          flag++;
45          return 5;
46      }

48      private int getTriesRemaining(){
49          return 2;
50      }


53  }
```

Listing C.4: Java code example of the control flow integrity countermeasure. Line 21 is removed in CFI2 in experiments

Bibliography

[1] Mehdi-Laurent Akkar, Louis Goubin, and Olivier Ly. *Automatic Integration of Counter-Measures Against Fault Injection Attacks*, 2003.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23068-7. doi: 10.1007/978-3-540-30080-9_7. URL http://dx.doi.org/10.1007/978-3-540-30080-9_7.

[3] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015. ISSN 1433-2779. doi: 10.1007/s10009-014-0361-y. URL http://dx.doi.org/10.1007/s10009-014-0361-y.

[4] Kristian Mikkel Thomsen Erik Sidelmann Jensen, Dennis Jakobsen and Christoffer Nduru. Java smart card security - an overview of fault injection attacks and countermeasures. Technical report, 2016.

[5] Houda Ferradi, Rémi Géraud, David Naccache, and Assia Tria. *When Organized Crime Applies Academic Results - A Forensic Analysis of an In-Card Listening Device*. https://eprint.iacr.org/2015/963.pdf. Visited 14. Dec 2015.

[6] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. *Sawja: Static Analysis Workshop for Java*. http://www.irisa.fr/celtique/pichardie/papers/foveoos10.pdf. Visited 10. Dec 2015.

[7] Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr Olesen, and René Rydhof Hansen. Study, formalisation, and analysis of dalvik bytecode. *Informal proceedings of The Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2012)*, 2012.

[8] Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter, and Koen De Bosschere. *Mitigating Smart Card Fault Injection with Link-Time Code Rewriting: A Feasibility Study*. http://fc13.ifca.ai/proc/7-2.pdf, 2013. Visited 2. Nov 2015.

[9] Sun Microsystems. Virtual machine specification, java card platform, version 2.2.2, 2006. URL http://download.oracle.com/otndocs/jcp/java_card_kit-2.2.2-fr-oth-JSpec/.

[10] D. Muraleedharan. *Modern Banking - Theory and Practice*. Prentice-Hall of India Pvt.Ltd, 2014. ISBN 9788120350328. URL http://www.amazon.com/Modern-Banking-Practice-D-Muraleedharan/dp/8120350324.

[11] Ahmadou A. Sere, Julien Iguchi-Cartigny, and Jean-Louis Lanet. *Evaluation of Countermeasures Against Fault Attacks on Smart Cards*. `http://www.sersc.org/journals/IJSIA/vol5_no2_2011/4.pdf`. Visited 3. Dec 2015.