Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing

Aymerick Savary[1,2], Marc Frappier[1], and Jean-Louis Lanet[2]

[1] University of Sherbrooke
[2] University of Limoges

Abstract. Java Card security is based on different elements among which the bytecode verifier is one of the most important. Finding vulnerabilities is a complex, tedious and error-prone task. In the case of the Java bytecode verifier, vulnerability tests are typically derived by hand. We propose a new approach to generate vulnerability test suites using model-based testing. Each instruction of the Java bytecode language is represented by an event of an Event-B machine, with a guard that denotes security conditions as defined in the virtual machine specification. We generate vulnerability tests by negating guards of events and generating traces with these faulty events using the ProB model checker. This approach has been applied to a subset of twelve instructions of the bytecode language and tested on five Java Card bytecode verifiers. Vulnerabilities have been found for each of them. We have developed a complete tool chain to support the approach and provide a proof of concept.

Keywords: Model Based Testing, Java Card bytecode Verifier, Vulnerability Testing, Security, Event-B.

1 Introduction

The Java Card technology [18] is a subset of the Java platform [16] that enables Java programs to run on resource constrained platforms like smart cards and other small devices. Security is an important concern in this platform, and it is ensured through various mechanisms *i.e.,* the firewall, the bytecode Verifier (BCV), the sharing mechanism... The firewall is in charge to provide segregation between different application providers. The sharing mechanism implements a security policy using an identity based protocol to allow information flows between different application providers. The BCV checks by static analysis that a Java Card program satisfies security constraints defined in the Java Virtual Machine specification (JVM). In the last version, 3.0.4 *Connected Edition* of the Java Card technology, the BCV is mandatory and must be executed on the smart card. This raises the issue of checking the correctness of the embedded BCV. Since a smart card has limited resources, developers may be tempted to optimize the BCV, possibly introducing subtle errors through complex optimization techniques. Testing such devices is a delicate and time consuming task.

E.B. Johnsen and L. Petre (Eds.): IFM 2013, LNCS 7940, pp. 223–237, 2013.

Thus, special care must be taken to ensure good coverage while minimizing the number of test cases, because testing such embedded systems is more laborious than stand-alone software systems.

BCV test cases are typically derived by hand. In this paper, we propose an approach to automate the generation of BCV test cases. We distinguish two classes of tests: i) conformance tests: they ensure that correct bytecode programs are indeed accepted by the BCV; ii) vulnerability tests: they ensure that incorrect Java bytecode programs are indeed rejected by the BCV. Detecting vulnerabilities is critical from a security point of view, because accepting incorrect programs may lead to attacks. These two classes require different test generation strategies; in this paper, we focus on the generation of vulnerability tests.

We adopt a model-based approach for the generation of vulnerability tests. This is an alternative to the proof and code generation method used by [5]. The idea is to model each function of the program (in this case each instruction of the language) by an event of an Event-B model. The event's guard represents the precondition of the instruction as defined in the JVM specification, which expresses the security constraints on an instruction. The event's action represents the result of executing the corresponding instruction. Since the JVM specification essentially addresses type checking, the Event-B specification abstracts from the actual value of bytecode variables and only models their types. To generate a test, we use the execution traces of this Event-B model, since each event in the trace denotes an instruction. To generate vulnerability tests, we modify the specification of an event to negate its guard, thus representing a violation of the JVM specification, in order to generate traces that denote invalid programs. This approach is modular, easily extensible, and it can reuse existing tools like ProB [15] to generate test cases.

The rest of this paper is structured as follows. Section 2 describes some security issues of Java-based smart cards. Section 3 describes our methodology for generating vulnerability tests. Section 4 reports on the application of our approach to five Java Card bytecode verifiers. Finally, we conclude with an appraisal of our approach and future work in Section 5.

2 Java Card Security Issues

The Java Card platform is a multi-application environment where critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses type verification, class loader and security managers to create private name spaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

To allow code to be loaded into the card after post-issuance raises security issues similar to those of web applets. An applet not built by a compiler (handmade bytecode) or modified after the compilation step may break the Java sandbox

model. Thus, the client must check that Java typing rules are preserved at the bytecode level. However, an attacker may attempt to build a bytecode program that confuses a return address with an object reference, thus allowing inspection and modification of critical memory values. The absence of pointer operators in the Java programming language reduces the number of programming errors. But it does not stop attempts to break security protections with unfair uses of pointers.

2.1 Logical Attacks in Smart Card

An attack can be carried using an ill–formed applet to obtain sensitive information stored in the card; for obtaining it, the applet will try to execute some illegal instructions to read and write in the smart card memory as explained in [11]. This can be accomplished by making a type confusion attack or by changing the control flow graph. Type confusion blurs the Java Card Runtime Environment to use reference to an object's instance as a value. In Java Card, references are mainly stored as 16 bit, i.e. the size of a short. This attack can be achieved by pushing a value and manipulating it as a reference. There are four methods to obtain a type confusion.

1. Input file manipulation. The goal is to modify the file after the compilation step to bypass the BCV. An on-card BCV will mitigate these attacks. Other BCVs are only partially embedded due to the smart card constraints.
2. Fault injection. This technique injects energy on the chip which is transformed into electric signals, which in turn can change values in memory or let the program behave differently by skipping instructions, inverting results and so on.
3. Shareable interfaces mechanisms abuse. To perform this attack, one creates two applets which communicate using the shareable interface mechanism. To create a type confusion, each of the applets use a different type of array to exchange data and are compiled separately. During the load phase, there is no way for the BCV to detect such a problem.
4. Transaction mechanisms abuse. The purpose of a transaction is to bundle a group of operations together. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, sometimes cards keep the references of objects allocated during transaction even after a roll back. Then, allocating a new object allows to point on the same memory segment with two references having two different types.

The first approach is the easiest way to perform logical attack against smart cards. The (3) and (4) are now correctly handled by recent smart cards. The second approach requires specific equipment, but its effects are exactly the same as the first attack and can be partially mitigated with dynamic run time type verification.

2.2 Java Card Byte Code Verifier

The BCV is a complex piece of software, and the algorithms to perform the verification were too expensive both in term of memory and computing requirements to be embedded in a Java Card except in the 3.0.4 *Connected Edition* version. In this version, the BCV is similar to the KVM verifier [17] where the idea is to separate the verification process in two parts: an off-card part, that computes a certificate, or "proof" that the code is correct with respect to the security policy, and an on-card part, that uses the certificate to verify the correctness of downloaded code. The "proof" generated is similar to the *StackMap* attribute used by the KVM, and contains the same kind of information. Due to the fact that no products are available for this platform, we focused on the 3.0.4 *Classic Edition* version, an evolution of the 2.2 version where the BCV is optional.

This section describes the Java byte code verification process that has to be performed. This verification should be performed for each package loaded and should reject the whole package if one of the tests fail. The full description of the verification can be found in [16], and a more detailed description, with appropriate discussions is given in [7]. This last description clarifies most of the unclear or ambiguous parts in the official JVM specification. A difference between Java and Java Card verification is that the verification has to be performed on Converted APplet (CAP) files for Java Card instead of class files. A CAP file is a tokenized and optimized version of a set of Java classes.

First, tests are performed on a CAP file when it is downloaded in order to ensure that the CAP file is well formed. Those tests do not analyze the code, but aim to check that the file is well structured. For example, it checks that no method is empty, or that mandatory parts of the file exist. For example, it is ensured that no final method is overridden, or that no class inherits from one of its subclasses. Moreover, in the case of Java Card, if one of the loaded classes already exists in the card, then the verification should fail.

Then, the type correctness of the program is verified. This verification is performed on a method basis, and has to be done for each method present in the package. When a method is invoked, a *frame* is created on top of the *Java virtual machine stack*. A frame contains the method's local variables and an *operand stack* which is used to store intermediate results during method execution. The size of the operand stack of a method is determined at compile time. JBC instructions play with these variables and the operand stack. A frame state denotes the value of the local variables and the operand stack.

Type checking ensures that no disallowed type conversions are performed. For example, an integer cannot be converted into an object reference, downcasting can only be performed using the `checkcast` instruction, and arguments provided to methods have to be of compatible types. Since the types of the local variables is not explicitly stored in the bytecode, they are derived by analysing the bytecode. This part of the verification is the most complicated one, and is both time and memory expensive. It requires computing the type of each variable and stack element for each instruction and each execution path. In order to make such verification possible, the verification is quite conservative on the programs

that are accepted. The standard bytecode verification algorithm only accepts programs where the type of each element in the stack and local variable is the same, whatever the path taken to reach an instruction. This also requires that the size of the stack is the same for each instruction for each path that can reach this instruction. Additionally, as every method defines the maximum size that the stack can take during execution, it is checked so that neither overflow or underflow can occur.

Here is an excerpt of the JVM specification for instruction `sload x`, which loads a short from the local variable identified by index `x` in the frame of a method invocation.

> **Stack**
> ... →
> ..., `value`
> **Description**
> *The index is an unsigned byte that must be a valid index into the local variables of the current frame [...]. The local variable at index must contain a short. The value in the local variable at index is pushed onto the operand stack.*

The description section provides the precondition and the postcondition of the instruction, while the stack section describes how the operand stack is modified by the instruction and the element required on the top of the stack before execution.

2.3 Verifying the Verifier

Cohen [2] has done a preliminary work on verifying the correctness and proposed a complete formalization for defensive JVM using ACL2. Each instruction in this model consists of operational semantics that describes its behaviors and also the static constraints that express the conditions needed to execute the instruction. Stata et Abadi [21] were the first to use typing rules to model the BCV. These rules precise the behavior of the instructions, describing the inputs, the execution context and all the postconditions of each instruction on the context. Freund et Mitchell [9] used the same framework to evaluate the object initialization considering only a minimum set of instructions. They added other Java features like classes, interfaces, arrays and exception in [10] and they proved the correctness of their type system.

Considering a set of important Java instructions, Qian [20] achieved one of the most complete works which proved the correct execution of a program by verifying its type system. He also proposed a proof for the verifier which is extracted from its formal model. In [6], a correct implementation of a BCV was explained by considering the verification problem as a data flow analysis and the executable was extracted using the Specware tool. Push *et al* [19] in the project named Bali used the Isabelle/HOL prover to define and verify the properties on subset of Java called μ-java. They formalized Qian's type system and its semantics. Nipkow in [13] has modeled a complete Java BCV using Isabelle.

His idea was to provide a generic proof for the verification algorithm and to instantiate it for a particular VM. The specific verification algorithm exploiting the *StackMap* attribute has been proved correct using its complete formalization and its proof in Isabelle [12].

In 1998 Gemplus demonstrated the correctness of Java Card optimizations availing B method [14]. Deutsche Telekom [3] employed a model checker (SMV) to demonstrate the Java Card verification algorithm which was realized using Linear Temporal Logic. A similar approach by applying the SMV model checker was used by Gemplus [4] to ensure whether the confidentiality of a shared data was preserved for a given applet using a causal dependency model. The first smart card using synthesized code from formal specifications was exhibited at Java One by Gemplus in 2002.

From all these studies, it is obvious that such a piece of standardized code and its implementation should be correct. However, Thales ITSEF [8] reported at the Common Criteria conference in 2010 a bug that allows a type confusion in a Java Card. The specification of a verifier changes very rarely, but Java Card is an exception with on-card verifiers. As high-level optimization is required, some differences may be expected.

There are very few implementations of verifiers that are publicly used. We assume that Oracle's verifier is the most commonly used, even if each smart card manufacturer has developed its own optimized version. Testing a BCV is a hard task. Static code analysis tool are used, but they are not easy to use due to the level of abstraction required. So there still exists an issue with both Java Card editions. The correctness of a particular implementation of the bytecode verifier needs at least a test suite or a methodology to check its correctness.

3 Methodology for Generating Vulnerability Tests

Our goal is to generate vulnerability tests for the BCV. We proceed as follows. A BCV vulnerability test is a faulty bytecode program. A bytecode program is a sequence of bytecode instructions. To generate a bytecode program, one can build a formal model of the bytecode language where each operation denotes a bytecode instruction. An execution trace of such a model then denotes a bytecode program. A faulty bytecode program contains an instruction which can be executed when its precondition is false. Thus, to generate a faulty bytecode program, one simply has to negate the precondition of an instruction and try to execute it in an execution trace. We use the Event-B notation to represent our formal model. In Event-B, operations are called "event", so we will use that term in the sequel.

In order to verify our approach, we have selected a subset of twelve instructions of the Java bytecode language (`aconst_null`, `pop`, `return`, `sadd`, `sconst_n1`, `sconst_0`, `sconst_1`, `sconst_2`, `sconst_3`, `sconst_4`, `sconst_5`, `sload`). These instructions manage the operand stack and the local variables. Our model can be used for testing stack overflows and underflows, type confusion on primitive types, for both local variables and stack elements.

For example, an accepted test could be the following sequence (where local variable at index 4 is a *short*): [`sload_4; sconst_2; sadd; return`].
A rejected test could be: [`sconst_2; aconst_null; sadd; return`].

3.1 The Formal Model

Figure 1 represents the variables and the invariants of the Event-B model. Variable *pc* denotes the index of the next instruction to be generated. The frame state of an instruction is represented by variables s and v, which respectively denote the operand stack and the local variables associated to each instruction of the bytecode program to be generated, whose length is given by constant *maxpc*. Thus, we store a copy of the "before" frame state for each instruction. It allows us to generate test cases for branching instructions. Branching instructions entails that an instruction can be reach from several execution paths. In a valid bytecode program, all frame states resulting from an execution path to an instruction must be type compatible, so that whatever path is taken, an instruction is always executed with valid types. Constant *maxstack* denotes the maximum size of the operand stack, which is determined during compile time. The size of the stack $s(i)$ is given by $z(i)$; this variable is necessary to generate faulty instructions for stack underflows. Variable *halt* is set to true when the program has reached a valid frame state for completing a method.

INVARIANTS

inv1 : $\mathtt{pc} \in 1 \mathinner{..} \mathtt{maxpc}$
inv2 : $\mathtt{s} \in 1 \mathinner{..} \mathtt{maxpc} \pfun (0 \mathinner{..} \mathtt{maxstack} - 1 \pfun \mathtt{TYPE})$
inv3 : $\mathtt{v} \in 1 \mathinner{..} \mathtt{maxpc} \pfun (1 \mathinner{..} \mathtt{maxlocalvar} \pfun \mathtt{TYPE})$
inv4 : $\mathtt{z} \in 1 \mathinner{..} \mathtt{maxpc} \pfun 0 \mathinner{..} \mathtt{maxstack}$
inv5 : $\mathtt{halt} \in \mathtt{BOOL}$
inv7 : $\mathtt{dom(s)} = \mathtt{dom(v)} \land \mathtt{dom(s)} = \mathtt{dom(z)} \land \mathtt{dom(v)} = \mathtt{dom(z)}$

Fig. 1. Invariants

Figure 2 represents the *Initialisation* event. The initial state of the machine contains, for instruction 1, an empty stack and some local variables defined by constant *initlocalvar*; the frame states of the other instructions are undefined.

The model of instruction `sload` is given in Figure3. Guards `grd1_t`, `grd2_t` and `grd3_t` represent the security conditions defined in the Java Specification [16]. Guards `grd1` and `grd2` have been added to control the test generation process. Guards with suffix "`_t`" will be negated to generate test cases; guards without suffix "`_t`" are never negated. Guard `grd1` ensures that no event can be executed after variable `halt` has been set to `TRUE`; variable `halt` is set to `TRUE` by instruction `return`, which ends each bytecode execution for a method. Guard `grd2` ensures that the length of execution traces does not exceed `maxpc`. Stack overflow is controlled by guard `grd1_t`. Guard `grd2_t` checks that the index is a valid index of the array of local variables. Guard `grd3_t` checks that the type of the local variable at the given index is a short. The event actions modify the

EVENTS
Initialisation
begin
act1 : pc := 1
act2 : s := {1 ↦ ∅}
act3 : v := {1 ↦ initlocalvar}
act4 : z := {1 ↦ 0}
act6 : halt := FALSE
end

Fig. 2. Initialisation

frame state of the next instruction. Thus, a short is pushed onto the stack (act2 and act3) and the local variables are left unchanged (act4).

EVENTS
Event *sload* ≙
any
index
where
grd1 : halt = FALSE
grd2 : pc < maxpc
grd1_t : z(pc) ≤ maxstack − 1
grd2_t : index ∈ 1 .. maxlocalvar
grd3_t : v(pc)(index) = short
then
act1 : pc := pc + 1
act2 : s := s ⩤ {pc + 1 ↦ s(pc) ⩤ {z(pc) ↦ short}}
act3 : z := z ⩤ {pc + 1 ↦ z(pc) + 1}
act4 : v := v ⩤ {pc + 1 ↦ v(pc)}
end

Fig. 3. sload event

3.2 The Test Generation Process

To obtain the test suite using MBT, the process is split into 3 steps as illustrated in Figure 4: test generation, concretization and execution. We have developed a tool for each step. Abstract test generation is performed by the Vulnerability Tests Generator (VTG). Then the XML2CAP tool translates these abstract tests to CAP files. Each CAP file is a concrete test. Finally TestOnPC and TestOnCard execute the tests on the off-card part and on-card part, respectively.

VTG, depicted in Figure 5, is the test generator. Compared to a traditional MBT approach, VTG includes a new step, faulty model derivation, between the model and the test generation. This new step generates a set of faulty models from the original model.

Tests can be split in 3 parts. The preamble leads the system under test (SUT) from the initial state to a state where it is possible to execute the body. The body is the execution of the tested behavior. Finally the postamble leads the SUT to a desired state.

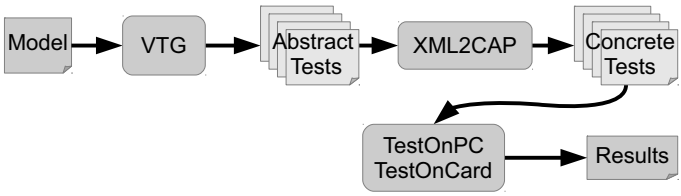

Fig. 4. Our MBT process

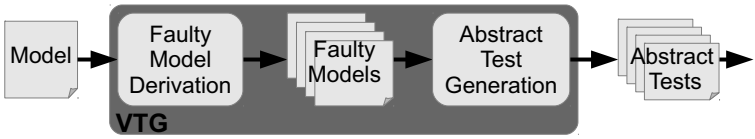

Fig. 5. VTG process

3.3 Faulty Model Derivation

Algorithm 1 describes the faulty model derivation process. Each generated faulty model contains only one faulty instruction, in order to ease the location of faults in the BCV. A faulty instruction is a negation of the JVM specification preconditions. Since there are several ways of negating a condition, several models are produced for a single faulty instruction. The faulty model contains a new state variable, `eut` (event under test) which ensures that a faulty instruction in executed only once in a test. Variable `eut` is initialized to `FALSE` and set to `TRUE` by the execution of the faulty instruction; a guard "`eut` = `FALSE`" is added to the faulty instruction so that it is executed only once.

To negate a guard g, the algorithm uses function $neg(g)$, which returns the set of negations of guard g. It is computed by recursively applying *derivation rules*. A negation g' of a guard g satisfies the following property: $g' \Rightarrow \neg g$. Thus, there are several possible negations g' for a guard g. The negations we consider are defined by derivation rules. The derivation rules necessary to rewrite the instructions of our subset of the Java bytecode language are presented in Figure 6. A rule has the following form: $neg(f) \rightsquigarrow \{f_1, \ldots, f_n\}$. Each f_i is a negation of f, and it may include a call to neg as a subformula. Thus, these rules are applied recursively until no more neg appear. Termination is ensured by (manually) checking that the rules decrease the height of the formula's abstract syntax tree. Completeness is manually checked by ensuring that $\neg f \Leftrightarrow f_1 \vee \ldots \vee f_n$. These are proved using the prover of the Rodin toolkit, which supports the Event-B method.

Input: m : *Event-B* model
Output: M' : set of *Event-B* model
for *each event* e *of* m **do**
  rewrite the guard of e into two guards:
    grd, the conjunction of all guards of e without suffix "`_t`";
    grd_t, the conjunction of all guards of e with suffix "`_t`";
end
for *each event* e *of* m **do**
  $e.RW := neg(grd\_t)$;
end
for *each event* e *of* m **do**
  for *each* rw *in* $e.RW$ **do**
    add a new model m' to M' such that
      $m' := m$;
      $m'.events := m'.events \cup \{e'\}$, where e' is defined as follows:
        $e' := e$;
        replace $e'.grd\_t$ by rw;
        add guard "`eut` = `FALSE`" to e';
        add action "`eut` := `TRUE`" to e';
  end
end

Algorithm 1. Faulty model derivation algorithm

Using derivation rules provides flexibility for controlling the faulty model generation process. For instance, rule 1 describes that a conjunction can be negated in three different ways: exactly one of the conjunct is false or both conjuncts are false. Some rules are also specific to a problem domain. For instance, to negate a formula $h(a) = b$, one may want to test two cases, instead of using rule 4 of Figure 6: i) the case where $h(a)$ is undefined (*i.e.,* $a \notin dom(h)$) and the case where $h(a)$ is defined. This would be represented by the following rule:

$$4'.\quad neg(h(a) = b) \rightsquigarrow \{a \notin dom(h),\ a \in dom(h) \wedge a \mapsto y \notin h\}$$

The application of derivation rules to a guard may generate a predicate which is unsatisfiable, or there may not exist a state reachable from the initial state that satisfies the generated predicate. These cases are detected during the abstract test generation step, which involves a model checker. In our tool, the user can check the list of generated predicates and delete those which are obviously not satisfiable, in order to speed up the test generation step.

As an example of applying derivation rules, consider instruction `sload`. Its guard to negate (*i.e.,* the conjunction of guards with suffix "`_t`" in Figure 3) is the following:

$$(z(pc) \leq maxstack - 1) \wedge (index \in 1\,..\,maxlocalvar) \wedge (v(pc)(index) = short)$$

Applying the rules of Figure 6, we obtain the following negations; elements identified in red highlight the modified part of the original guard.

1. $neg(p_1 \wedge p_2) \rightsquigarrow \{neg(p_1) \wedge p_2,\ p_1 \wedge neg(p_2),\ neg(p_1) \wedge neg(p_2)\}$
2. $neg(i_1 \leq i_2) \rightsquigarrow \{i_1 > i_2\}$
3. $neg(i_1 \geq i_2) \rightsquigarrow \{i_1 < i_2\}$
4. $neg(i_1 = i_2) \rightsquigarrow \{i_1 \neq i_2\}$
5. $neg(a \in B) \rightsquigarrow \{a \notin B\}$

Fig. 6. Relevant derivation rules

1. $z(pc) > maxstack - 1 \wedge index \in 1\,..\,maxlocalvar \wedge v(pc)(index) = short$
2. $z(pc) \leq maxstack - 1 \wedge index \notin 1\,..\,maxlocalvar \wedge v(pc)(index) = short$
3. $z(pc) > maxstack - 1 \wedge index \notin 1\,..\,maxlocalvar \wedge v(pc)(index) = short$
4. $z(pc) \leq maxstack - 1 \wedge index \in 1\,..\,maxlocalvar \wedge v(pc)(index) \neq short$
5. $z(pc) > maxstack - 1 \wedge index \in 1\,..\,maxlocalvar \wedge v(pc)(index) \neq short$
6. $z(pc) \leq maxstack - 1 \wedge index \notin 1\,..\,maxlocalvar \wedge v(pc)(index) \neq short$
7. $z(pc) > maxstack - 1 \wedge index \notin 1\,..\,maxlocalvar \wedge v(pc)(index) \neq short$

Negations 2, 3, 6 and 7 are unsatisfiable, because of the conjunction $index \notin 1\,..\,maxlocalvar \wedge v(pc)(index) = short$. When $index \notin 1\,..\,maxlocalvar$ holds, expression $v(pc)(index)$ is undefined. Figure 7 illustrates a faulty instruction obtained with negation 5.

```
EVENTS
Event evt_sload_11_24_EUT ≙
    any
        ...
    where
        grd : halt = FALSE ∧ pc < maxpc
        grd_t : z(pc) > maxstack − 1 ∧ index ∈ 1 .. maxlocalvar ∧
                ¬v(pc)(index) = short
        grd_EUT : eut = FALSE
    then
        ...
        act_EUT : eut := TRUE
    end
```

Fig. 7. `evt_sload_11_24_EUT` event

3.4 Abstract Tests Generation

With these new models, we generate tests. We use ProB [15] to find traces containing the preamble, the body and the postamble. This search is driven by two parameters. First we specify the depth. This parameter represents the maximum length of the traces *i.e.,* the maximum number of events for a test. The second parameter is a predicate that each trace must satisfy. Presence of the EUT and final state are represented by the predicate $eut = TRUE \wedge halt = TRUE$. Other parameters must be specified; the reader is referred to the ProB website [1] for more details.

For each model, we only generate a subset of possible traces containing the EUT. For each transition, when several solutions can satisfy the guard of an event, only one solution is used. The first solution found for the model of Figure 7 is :

- preamble: `INITIALISATION; aconst_null; aconst_null; aconst_null;`
- body: `evt_sload_11_24_EUT`;
- postamble: `return` .

At the initialization, the local variables contain two references and the maximum size of stack is equal to 3. The execution of three events *aconst_null* fills the stack. In this state, the faulty event can be executed. Finally the return event leads the machine to the end of the test.

3.5 Concrete Tests

A concrete test is a CAP file which contains a class with one method. This method contains the instruction trace generated with ProB. The XML2CAP tool generates one CAP file for each trace. It computes, among other things, the maximum stack size. Because several other informations are required for composing a CAP file, we use a predetermined CAP file which is completed with the generated method. The local variables are fixed in the predetermined CAP file and in the abstract model.

4 Evaluation

Tests Computation. Our experiments have been performed on a MacBook Pro with a 2,3 GHz Core i5 dual-core processor, 8GB of RAM and a 5400 $t.m^{-1}$ hard disk with 8MB of cache. Each measure has been made three times and we provide the average.

The first step is the negation of guards. It processes four distinct guards. It generates sixteen negations in less than one second. Eight of the negations are unreachable and we only keep the eight reachable models. The second step is the model derivation. We produce eighteen new models in twenty seconds. The last step is the abstract test generation. The results obtained vary depending on the search depth.

Table 1 represents the results we obtain for the generation of abstract tests. The first column represent the depth parameter. The penultimate line of this table represents the results we obtain if we do not remove the unreachable negations. In the last line we take the shortest depth for each model. The second and third columns represent respectively the time and the number of tests extracted. The next column represent the time taken in average to generate one test. The penultimate column provides the percentage of models which can produce at least one test for the given depth (the depth may not be large enough to generate a test). The last column represents the test coverage with respect to a test plan manually derived by an expert. In this manual test plan, the expert has

identified for each instruction a number of test cases. The test cases considered by our approach depend on the derivation rules. We have computed the number of manual test cases that our approach can reproduce using the derivation rules. Because we only work on a subset of the Java Card instruction set, the manual test cases which involve other instructions could not be reproduced; thus the last column is never equal to 100%.

Table 1. Abstract test generation evaluation

Depth	Time	Nb of tests	Speed ($sec/test$)	Model coverage	Manual plan coverage
3	1min30	30	3,0	13%	12%
4	12min30	432	1,7	33%	31%
5	1h30	10133	0,5	100%	93%
5 Full	2h30	10133	0,9	65%	93%
*	1h05	7318	0,5	100%	93%

Smart Card Execution. After the execution of the vulnerability test suites, we classify the results as: i) accepted and correctly executed CAP file, ii) rejected CAP file, and iii) accepted CAP but rejected executions. An accepted CAP means that the embedded load phase verifications do not detect the ill formed file. An accepted CAP but rejected during execution detects the presence of a run time check. The test suites have been evaluated on five different smart cards from two different smart card manufacturers ($\{a, e\}, \{b, c, d\}$). Cards a, b, c are Java Card 2.1 while d and e are Java Card 2.2 standard.

Card d allows the execution of all tests. Thus, it is possible to generate stack over and under flow. It does not mean that the vulnerabilities can be exploited. To obtain an executable attack, one often needs several vulnerabilities. For example, if the card does not check the local variable it offers the possibility to change the return address. But the return address can be protected by an integrity check. With our test results, one may be able to characterize a given implementation and then provide information to set up an attack.

For the other cards, some tests fail (the card become mute) during the execution or during the load. On card a, we have been able to find a chain of vulnerabilities that allows us to execute a shell code on the card.

Overall Effort for the Complete Process. We estimate that it would take a fresh person about 25 hours to build the formal model of the subset of twelve instructions. We could not precisely measure this effort, because our model is the result of several iterations on different subsets of similar bytecode languages (for instance, our first experiments were conducted on the language of Freund and Mitchell [10]). The manual removal of unreachable guards takes about 1.5 hour. The model derivation step does not require any human intervention. We then must choose appropriate parameters for the abstract test generator step. This will only take a few minutes. Then we launch the abstract test generator.

For a full coverage, this step takes 1.5 hour. Finally we concretize the tests in a few minutes. Overall, the full process for the twelve instruction subset takes about a week.

It took eight person-months to develop the complete toolchain. Many parts of this toolchain are re-usable. Only the concretization and the execution tools must be adapted for a new problem.

Manually writing a test suite for the BCV is tedious. It took us one week to manually derive a test suite for our subset of twelve instructions. Moreover, this test suite contains only one test per test case. Our solution can produce all possible tests in roughly the same time; we have controlled the number of tests using a timeout. Our vulnerability coverage is slightly less, but our tests are more complex and they test vulnerabilities in many contexts. When the full bytecode instruction set will be tackled, we expect increased productivity gains, because automation will easily generate a greater variety of contexts for testing an instruction.

5 Conclusion and Future Work

We have proposed a new methodology to generate vulnerability tests and developed several tools supporting it. Our method is based on using a formal model of the system under test which represents security constraints. A standard MBT approach is applied and we obtain a set of vulnerability tests. We applied our methodology to the testing of five Java smart card BCV. We have discovered vulnerabilities in all of them. Our approach can be used by a certification authority or an evaluation center in order to set up vulnerability analysis. This would ease the characterization of the embedded software.

The Java Card byte code verifier is a key component in the security of Java-based smart cards and finding weaknesses is of prime importance. Our experiment with a subset of the Java Card instruction set constitutes a proof of concept. We plan to apply our methodology to the complete Java Card instruction set, including the type lattice, the subroutine mechanism and the exception mechanism.

Our methodology is generic and can be applied to other security components. We have started to model the payment protocol EMV, which includes cryptographic primitives, in other to generate vulnerability tests.

References

1. `http://www.stups.uni-duesseldorf.de/ProB`
2. `http://www.computationallogic.com/software/djvm/`
3. Basin, D., Friedrich, S., Posegga, J., Vogt, H.: Java bytecode verification by model checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 491–494. Springer, Heidelberg (1999)
4. Bieber, P., Cazin, J., Girard, P., Lanet, J.L., Wiels, V., Zanon, G.: Checking secure interactions of smart card applets: Extended version. Journal of Computer Security 10(4), 369–398 (2002)

5. Casset, L.: Development of an embedded verifier for java card byte code using formal methods. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 290–309. Springer, Heidelberg (2002)
6. Coglio, A., Goldberg, A., Qian, Z.: Toward a provably-correct implementation of the JVM bytecode verifier. In: Proc. OOPSLA 1998 Workshop on Formal Underpinnings of Java, pp. 403–410 (1998)
7. Doyon, S.: On the security of Java: The Java Bytecode Verifier. Master's thesis, Université Laval, Québec City, Canada (1999)
8. Faugeron, E.: How to hoax an off-card verifier. In: e-Smart, Sophia Antipolis, France, September 21-24. Strategies Telecoms & Multimedia, pp. 310–328 (2010)
9. Freund, S.N., Mitchell, J.C.: A type system for object initialization in the Java bytecode language. In: Freeman-Benson, B.N., Chambers, C. (eds.) ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, pp. 310–327. ACM Press (1998)
10. Freund, S.N., Mitchell, J.C.: A formal framework for the Java bytecode language and verifier. In: Hailpern, B., Northrop, L.M., Berman, A.M. (eds.) ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, pp. 147–166. ACM Press (1999)
11. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology 6, 343–351 (2010)
12. Klein, G., Nipkow, T.: Verified lightweight bytecode verification. Concurrency and Computation: Practice and Experience 13(13), 1133–1151 (2001)
13. Klein, G., Nipkow, T.: Verified bytecode verifiers. Theor. Comput. Sci. 3(298), 583–626 (2003)
14. Lanet, J.L., Requet, A.: Formal proof of smart card applets correctness. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 14–16. Springer, Heidelberg (2000)
15. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer 10(2), 185–203 (2008)
16. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
17. Sun Microsystems: Connected, limited device configuration, specification 1.0a, Java 2 platform micro edition (2000)
18. Sun Microsystems: Virtual machine specification Java Card platform (May 2009), `http://www.oracle.com`
19. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in isabelle/HOL. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 89–103. Springer, Heidelberg (1999)
20. Qian, Z.: A formal specification of java-TM virtual machine instructions for objects, methods and subroutines. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 271–312. Springer, Heidelberg (1999)
21. Stata, R., Abadi, M.: A type system for Java bytecode subroutines. In: MacQueen, D.B., Cardelli, L. (eds.) Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, San Diego, CA, USA, January 19-21, pp. 149–160. ACM (1998)