Tool-Assisted Specification and Verification of the JavaCard Platform

Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa*

INRIA Sophia-Antipolis, France

{Gilles.Barthe,Pierre.Courtieu,Guillaume.Dufay,Simao.DeSousa}@inria.fr

Abstract. Bytecode verification is one of the key security functions of the JavaCard architecture. Its correctness is often cast relatively to a defensive virtual machine that performs checks at run-time, and an offensive one that does not, and can be summarized as stating that the two machines coincide on programs that pass bytecode verification. We review the process of establishing such a correctness statement in a proof assistant, and focus in particular on the problem of automating the construction of an offensive virtual machine and a bytecode verifier from a defensive machine.

1 Introduction

Smartcards are small devices designed to store and process confidential data, and can act as enabling tokens to provide secure access to applications and services. They are increasingly being used by commercial and governmental organizations as bankcards, e-purses, SIM cards in mobile phones, e-IDs, *etc*, and are expected to play a key role in enforcing trust and confidence in e-payment and e-government [22].

Open platform smartcards are new generation smartcards with increased flexibility. Such smartcards:

- integrate on-board a virtual machine that abstracts away from any hardware and operating system specifics so that smartcard applications, or *applets*, can be programmed in a high-level language (before being compiled and loaded onto the card);
- are multi-applications, in that several applets can coexist and communicate (securely) on the same card, and support post-issuance, in that new applets may be loaded onto already deployed smartcards.

The flexibility of open platform smartcards is at the same time a major asset and a major obstacle to their deployment. On the one hand, writing applets

---

* On leave from University of Beira Interior-Portugal, and partially supported by the Portuguese research grant sfrh/bd/790/2000.

in a high-level language reduces the cost and time to market new applications, and the possibility of running several applets on a single card opens the way for novel applications. On the other hand, such smartcards introduce the possibility to load malicious applets that exploit weaknesses of the platform to read, modify or erase confidential data of other applets, or simply to block the card.

JavaCard and the JavaCard Platform. JavaCard [26] is the standard programming language for multi-applications smartcards. Launched in 1996 by Sun, JavaCard brings to the resource-constrained environment of open platform smartcards many benefits of Java: object-orientation, safety through typing, portability. Here is a brief description of JavaCard and of its differences with Java:

- JavaCard applets are written in a subset of the Java language that omits some items such as large datatypes, arrays of arrays, finalization or threads. Furthermore, JavaCard applets use the JavaCard API, which defines and constrains the interface between an applet and its environment on the card;
- JavaCard applets are compiled down to class files, and then converted to the *CAP file* format. Conversion involves a number of optimizations suggested by smartcard constraints, e.g. names are replaced by tokens and class files are merged together on a package basis;
- compiled JavaCard applets go through a bytecode verifier BCV which checks that applets are correctly formed and well-typed, and that they do not attempt to perform malicious operations during their execution, see [33]. Until recently, applets were verified off-card and, in case of a successful verification, signed and loaded on-card. However, on-card bytecode verification, which circumscribes the trusted computing base to the smartcard, should come to play a prominent role in the future [13, 14, 34];
- JavaCard applets are executed by the JavaCard Runtime Environment JCRE. The JCRE contains the JavaCard Virtual Machine JCVM, provides support for the JavaCard API and invokes the services of a native layer to perform low-level tasks such as communication management. The JCVM is a stack-based abstract machine as the Java Virtual Machine JVM but some features, like the dynamic class loading mechanism of the JVM, are unsupported;
- JavaCard claims some features that are not present, at least not in the same form, in Java: for example, JavaCard 2.1. introduced firewalls to ensure applet isolation, and JavaCard 2.2. introduced logical channels and a simplified form of remote method invokation.

Further technical information on JavaCard may be found in publicly available specifications, white papers and books [16, 43–46].

Formal methods for smartcards. It is commonly agreed that a high degree of reliability is required if smartcards are to gain a widespread acceptance as trusted computing devices, and that the highest degree of reliability can only be achieved by the use of formal methods to formally design, test and verify smartcard platforms and applications. This view is also present in the Common Criteria [21], that define a unified scheme for evaluating the security of IT products, and impose that evaluations at the highest levels, i.e. from EAL5 to EAL7, rely heavily on formal methods.

There are many facets to formal verification for smartcards, each of which comes with its own objectives and techniques. For example:

- program verification at source level may help developers to ensure that their applets are correct. One definite advantage of such a scenario is that the developer (or an expert in formal methods working with him) can provide a formal specification of the program, and then check/prove that the applet is correct with respect to its specification. There is a wide range of formalisms to express such specifications, such as predicate, temporal and dynamic logics or interface specification languages [27]. The corresponding verification techniques include run-time checking and unit testing [17], static checking [32], model-checking [19], and interactive theorem proving [9];
- program verification at bytecode level may help to check that it is safe to load and execute an applet whose origin is unknown or doubtful. As one cannot expect code consumers to engage in program verification, it is important that this kind of verification uses automatic techniques such as static analyses and model-checking [11, 39], or relies on a scheme which delegates the verification tasks to the code producer [36].

Platform verification, which is the focus of our work, aims at establishing that a platform for smartcard is correct in the sense that it adheres to some acceptable security policy. Such a security policy is usually formulated as a set of constraints like “a reference cannot be cast to an integer”, or “the virtual machine cannot jump to an arbitrary address”, and in the case of the JavaCard platform should include constraints such as type safety and applet isolation. The purpose of platform verification is to prove formally that these constraints are enforced by well-identified security functions. One important such security function is bytecode verification (it guarantees properties such as type correctness, code containment, proper object initialization, absence of stack overflows); proving its correctness is a prerequisite for Common Criteria evaluations at highest levels, and our starting point.

The correctness of bytecode verification is stated as a correspondence between two flavors of virtual machines [15, 30, 40]:

- a defensive JCVM, which manipulates typed values and performs structural and type checking at run-time. The defensive JCVM is close to Sun’s specification, but is inefficient in terms of execution speed and memory consumption;
- an offensive JCVM, which manipulates untyped values and relies on successful bytecode verification to eliminate structural and type checking at run-time. The offensive JCVM is close to an implementation;

The correctness statement can be summarised by the slogan: “the offensive and defensive JCVMs coincide on programs that pass bytecode verification”. Formulating precisely and proving the statement requires:

– modeling the defensive and offensive JCVMs and formalizing the notion of correspondence between the two virtual machines in terms of an abstraction function α_{do} that maps states of the defensive machine to states of the offensive machine;
– showing that both machines coincide on programs whose execution on the defensive JCVM does not raise a type error;
– modeling the BCV as a dataflow analysis of an abstract JCVM that only manipulates types and showing that the defensive JCVM does not raise a type error for programs that pass bytecode verification.

In the context of Common Criteria evaluations at the highest levels, such models and proofs are required to be formal and are best developed with the help of proof assistants.

CertiCartes [7, 8] is an in-depth feasibility in proving the correctness of bytecode verification for JavaCard 2.1, using the proof assistant Coq [18]. Currently CertiCartes contains executable specifications of the three JCVMs (defensive, offensive and abstract) and of the BCV, and a proof of the correctness of the BCV. The Coq development is structured in two separate modules:

1. the first module JCVM includes the construction of three virtual machines and their cross-validation. This development represents the bulk of the effort behind CertiCartes;
2. the second module BCV includes the construction and validation of the bytecode verifier. This development is largely independent from the definition of the virtual machines, which are taken as parameters in the construction of the BCV.

An external evaluation by Gemplus Research Labs [29] suggests that CertiCartes models the JavaCard platform at a reasonable level of detail, and proposes some concrete steps to make CertiCartes amenable to certification. It is not an avenue that we are intent on pursuing, in particular because the amount of work involved is considerable: [24] reports that six man years were necessary to carry a formalization of JavaCard 2.1 in full detail.

Assessment. Our experiment with CertiCartes corroborates the common views that machine checked verification of standard programming languages is becoming viable but sorely lacks of accurate tool support for specifying and proving in the large. What seems more interesting of CertiCartes is that it validates an original methodology which is well-suited for reasoning about typed low-level languages, and which can be applied to novel type systems for Java(Card) [4, 23, 31, 42]. Furthermore, our work suggested that it is possible to develop appropriate tool support for this methodology, and in particular for the construction of the offensive and abstract JCVM from the defensive one, and for their cross-validation.

Jakarta [6] is an experimental toolset for specifying and reasoning about the JavaCard Platform. The overall objective is to provide:

– a front-end, independent from a particular proof assistant, for describing virtual machines and subjecting such descriptions to transformations such as abstractions, refinements and optimizations;
– a compiler that translates these specifications in a format usable by proof assistants, and mechanisms for proving the correctness of these transformations in such proof assistants.

Our current focus is to provide tool support for the construction of an offensive and abstract JCVM from a description of a defensive JCVM, and for their cross-validation in a proof assistant. It may appear to be an over-specific goal but the defensive JCVM has many instantiations that are determined by the property to be enforced statically through typing. These properties typically relate to object initialization, information flow, resource control, locks, *etc*—we shall return to this point in the conclusion. Nevertheless type safety is the prime example of property to be enforced, and the running example of this paper.

In what follows, we shall: describe the methodology behind CertiCartes; explain how Jakarta has been used for extracting the offensive and abstract JCVMs from the defensive one; summarize ongoing work on automating the cross-validation between the three virtual machines.

2 An overview of CertiCartes

As indicated in the introduction, CertiCartes is structured in two separate modules: a first module JCVM, which includes the construction of three virtual machines and their cross-validation, and a second module BCV, which includes the construction and validation of the BCV from the abstract JCVM. The next paragraphs briefly describe the contents of CertiCartes; [7, 8] provide further details.

2.1 Module JCVM

Modeling programs. Programs are formalized in a neutral mathematical style based on (first-order, non-dependent) datatypes and record types, and the corresponding machinery: case analysis and structural recursion for datatypes, formation and selection for record types. For example, programs are represented by the record type:

```
Record jcprogram := {
  interfaces : (list Interface);
  classes    : (list Class);
  methods    : (list Method)
}.
```

where the types `Interface`, `Class` and `Method` are themselves defined as record types. For simplicity, we only deal with closed programs hence (relevant parts of) the packages `java.lang` and `javacard.framework` are an integral part of programs.

Modeling memory. Memory is modeled in a similar way. For example, states are formalized as a record consisting of the heap (containing the objects created during execution), the static fields image (containing static fields of classes) and a stack of frames (environments for executing methods). Formally:

```
Record state := {
  hp : heap;
  sh : sheap;
  st : stack
}.
```

In order to account for abrupt termination (that may arise because of uncaught exceptions or because the program being executed is ill-formed), we also introduce a type of return states, defined as a sum type:

```
Inductive rstate :=
  Normal   :              state → rstate |
  Abnormal : exception → state → rstate.
```

The definition of (return) states and of all components of the memory model are parameterized by a generic type of values, which we leave unspecified. The memory model will later be instantiated to the defensive, offensive and abstract JCVMs. The following table gives the notion of value attached to each JCVM:

Virtual Machine	Defensive	Offensive	Abstract
Value	(Type,Number)	Number	Type

Modeling the defensive JCVM. First, we instantiate the memory model with a notion of typed value. We thus obtain a type of defensive states `dstate` and return defensive states `rdstate`.

Then, we model the defensive semantics of each JavaCard bytecode `b` as a function `dexec_b: dstate → rdstate`. Typically, the function `dexec_b` extracts values from the state, performs type verification on these values, and extends/updates the state with the results of executing the bytecode.

Finally, one-step defensive execution is modeled as a function `dexec: dstate → rdstate` which inspects the state to extracts the JavaCard bytecode `b` to be executed and then calls the corresponding function `dexec_b`.

Modeling the offensive JCVM. First, we instantiate the memory model with a notion of untyped value so as to obtain a type of offensive states `ostate` and return offensive states `rostate`. Then, one-step offensive execution `oexec: ostate → rostate` is modeled in the same way as one-step defensive execution, but all verifications and operations related to typing are removed.

Modeling the abstract JCVM. The abstract JCVM that is used for bytecode verification operates at the level of types. Formally, we define the type `astate` of abstract states as a record type with two fields: an abstract frame and an abstract static fields image (which are obtained by instantiating values to types in the generic notions of frame and of static fields image). The memory model is thus simplified because: (1) the abstract JCVM operates on a method per method basis, so only one frame is needed; (2) the type of objects is stored in the operand stack so the heap is not needed.

One-step abstract execution is non-deterministic because branching instructions lead to different program points and hence different states. We use a list to collect all return states. Therefore one-step abstract execution is modeled as a function `aexec: astate → (list rastate)`, where `astate` is the type of return abstract states.

Cross-validation of the JCVMs is a prerequisite for establishing the correctness of the BCV, and involves showing that the offensive and defensive JCVMs coincide on programs that do not raise a type error when executed on the defensive JCVM; and that every program that raises a type error when executed on the defensive JCVM also raises a typing error when executed with the abstract JCVM. These statements are nicely expressed in terms of abstract interpretation. Indeed, the offensive and abstract JCVMs can be viewed as abstractions of the defensive JCVM. Likewise, the correctness statements can be viewed as expressing that the offensive and abstract JCVMs are sound abstractions of the defensive JCVM. Formally, we define two abstractions functions:

```
alpha_do : dstate → ostate
alpha_da : dstate → astate
```

together with their lifting to return states:

```
alpha_do_rs : rdstate → rostate
alpha_da_rs : rdstate → rastate
```

and prove that, under suitable constraints, the diagrams of Figures 1 and 2 commute (where the arrow on the right-hand side of Figure 2 means that the abstraction of the return defensive state is, up to subtyping, an element of the list of return abstract states).

$$
\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_do}"] & \texttt{rdstate} \arrow[d, "\texttt{alpha\_do\_rs}"] \\
\texttt{ostate} \arrow[r, "\texttt{oexec}"] & \texttt{rostate}
\end{tikzcd}
$$

Fig. 1. COMMUTATIVE DIAGRAM OF DEFENSIVE AND OFFENSIVE EXECUTION

$$\begin{tikzcd}
\texttt{dstate} \arrow[rr, "\texttt{dexec}"] \arrow[d, "\texttt{alpha_da}"] & & \texttt{rdstate} \arrow[d, "\preceq\texttt{alpha_da_rs}"] \\
\texttt{astate} \arrow[rr, "\texttt{aexec}"] & & (\texttt{list rastate})
\end{tikzcd}$$

Fig. 2. Commutative diagram of defensive and abstract execution

The commutation of both diagrams is proved by case analyses, first on the bytecode to be executed and second on the state of the defensive virtual machine, and finally by equational reasoning.

Summary The module JCVM represents most of the effort behind CertiCartes. However, the construction of the offensive and abstract JCVMs relies on abstract interpretation and can be automated. Likewise, the cross-validation of the JCVM relies on case analyses and equational reasoning and can also be automated.

2.2 Module BCV

The module BCV constructs a certified bytecode verifier from an abstract virtual machine using a formalization of Kildall's algorithm that is similar to the one presented in [28, 37]. Our formalization of Kildall's algorithm takes as input:

- a type `astate` of abstract states and a well-founded relation `lt_astate` over `astate`;
- an abstract virtual machine `aexec: astate` → `(list astate)` which is monotone w.r.t. `lt_astate`;

and produces as output:

- an executable function `bcv: jcprogram` → `bool`;
- a proof that a program `p` such that `bcv p = true` does not raise any typing error when executed on the abstract virtual machine `aexec`.

Additionally, the module BCV contains a justification of method-by-method verification, i.e. a proof that a program is well-typed if all its methods are. Together with the proofs of the commuting diagrams, these results ensure that bytecode verification is correct.

Summary The module BCV is largely independent from the specifics of the virtual machine under consideration, and hence could be applied to other settings with minimal effort.

3 Constructing certified abstractions with Jakarta

The objective of Jakarta is to provide an interactive environment for transforming virtual machines and for verifying the correctness of these transformations in proof assistants. Examples of such transformations include refinements and optimizations, but to date our work focuses on abstractions. In what follows, we present the abstraction mechanism of Jakarta, and describe how it has been used to extract the offensive and abstract JCVMs from the defensive JCVM.

3.1 Current Focus

The current focus with Jakarta is to provide automatic support for the construction and cross-validation of the virtual machines for an arbitrary property P. Concretely, we want the Jakarta toolset to take as input:

- a P-defensive virtual machine `dexec: dstate → dstate` that manipulates typed values (the notion of type being determined by P) and enforces P through run-time type-checks;
- two abstraction functions:

  ```
  alpha_da : dstate → astate
  alpha_do : dstate → ostate
  ```

  where `astate` and `ostate` respectively denote some suitably defined types of abstract and offensive states;
- transformation scripts which provide instructions on how to conduct the abstraction of the P-defensive virtual machine into a P-abstract (resp. P-offensive) one;

and produces as output a P-offensive and a P-abstract virtual machines:

```
oexec : ostate → orstate
aexec : astate → (list arstate)
```

together with the proofs that they constitute sound transformations of the P-defensive virtual machine.

3.2 An overview of Jakarta

The Jakarta Specification Language JSL is a small language for describing virtual machines in a neutral mathematical style. JSL is a polymorphically typed language whose execution model is based on term rewriting [3, 10].

A JSL theory is given by a set of declarations that introduce first-order, polymorphic datatypes and their constructors, and by a set of function definitions that introduce function symbols and fix their computational meaning via conditional rewrite rules of the form:

$$\mathtt{l}_1 \to \mathtt{r}_1, \ldots, \mathtt{l}_n \to \mathtt{r}_n \Rightarrow \mathtt{g} \to \mathtt{d}$$

A specificity of JSL is to require that the right-hand sides of the conditions, i.e the $\mathtt{r}_i$ s, should be patterns with fresh variables (recall that a pattern is a linear term built from variables and constructors). Such a format of rewriting seems unusual but is closely related to pattern-matching in functional programming and proof assistants. Figure 3 gives the JSL specification of the `ifnull` bytecode.

```
data valu_prim = VReturnAddress nat | VBoolean z |
                  VByte z| VShort z | VInt z.

data valu_ref = VRef_null | VRef_array type0 heap_idx |
                VRef_instance cap_class_idx heap_idx |
                VRef_interface cap_interf_idx heap_idx .

data valu = VPrim valu_prim | VRef valu_ref.

data list 'a = Nil | Cons 'a (list 'a).

data exc 'a = Value 'a | Error.

function  ifnull  : bytecode_idx → dstate → rdstate :=
  (stack_f state) → Nil
     ⇒ (ifnull branch state) → (abortCode State_error state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VPrim v0)
     ⇒ (ifnull branch state) → (abortCode Type_error state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VRef vr),
  (res_null vr) → True
     ⇒ (ifnull branch state)
         → (update_frame (update_pc branch
                            (update_opstack (tail (opstack h)) h)) state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VRef vr),
  (res_null vr) → False
     ⇒ (ifnull branch state)
         → (update_frame (update_pc (S (p_count h))
                        (update_opstack (tail (opstack h)) h)) state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → Error
     ⇒ (ifnull branch state) → (abortCode Opstack_error state).
```

Fig. 3. Example of JSL specification

The Jakarta Prover Interface provides a means to translate JSL specifications to proof assistants and theorem provers. Currently, we support translations from JSL to Coq, Isabelle [38], PVS [41] and Spike [12]. These translations may produce programs which are considered ill-formed by the target system (proof assistants often reject partial, non-deterministic or non-terminating functions), but it has not turned out to be a problem in practice.

Furthermore, we provide a translation from Coq to JSL so as to reuse existing Coq developments and a translation from JSL to OCaml [35] for an efficient execution of JSL specifications.

The Jakarta Transformation Kit JTK provides a means to perform automatic abstractions on JSL specifications, and is described in more detail in the next subsections. Finally, the *Jakarta Automation Kit* aims at facilitating the process of cross-validating virtual machines in proof assistants, and is discussed in Subsection 3.6.

3.3 The Jakarta Transformation Kit

Principles The abstraction $\hat{\texttt{S}}$ of a JSL theory `S` is constructed in three successive steps:

1. the user provides for every datatype `T` of `S` a corresponding abstract datatype $\hat{\texttt{T}}$ together with an abstraction function $\alpha_{\texttt{T}}$ from `T` to $\hat{\texttt{T}}$; abstraction functions are written in JSL, but are only allowed to use unconditional rewriting;
2. the user builds a transformation script that guides the abstraction process in places where automatic procedures are inaccurate; the purpose and effect of this script is detailed below.
3. based on the inputs of steps 1 and 2 above, the JTK constructs automatically for every function $\texttt{f:}\ \texttt{T}_1 \rightarrow \ldots \rightarrow \texttt{T}_n \rightarrow \texttt{U}$ of `S` its abstract counterpart $\hat{\texttt{f}}\texttt{:}\hat{\texttt{T}}_1 \rightarrow \ldots \rightarrow \hat{\texttt{T}}_n \rightarrow \hat{\texttt{U}}$. Unless indicated otherwise by the abstraction script, the function $\hat{\texttt{f}}$ is constructed by a syntactic translation on the rewrite rules defining `f`, followed by a cleaning phase where vacuous conditions and rules are removed.

Most of the abstraction process is handled automatically but there are however situations where user interaction is required. It is the purpose of the transformation script to facilitate such interactions through a carefully designed set of commands.

Commands fall into three categories:

discarding commands specify which patterns in expressions are unwanted and must be deleted. Discarding commands are used for example to remove all verifications and operations related to typing from the the offensive JCVM;

substitution commands specify coercions and substitutions that should be used during the translation of rewrite rules. Substitution commands are used for example to replace dynamic lookup by static lookup in the abstract semantics of method invokation;

post-processing commands deal with functions which may have become non-deterministic or non-total during abstraction. Post-processing commands are used for example to collect all possible return states into lists during the construction of the abstract JCVM.

Each command is of the general form:

$$<\!command\!> ::= <\!action\!>\ <\!target\!>list\ [\textbf{by}\ <\!expr\!>]\ [\textbf{in}\ <\!scope\!>list]$$

- The optional field *scope* determines in which part of the JSL specification the action must be performed; by default the scope of a command is the whole specification.
- The field *target* specifies which syntactic compounds trigger the action.
- The optional field *expr* is used in substitution commands to specify the syntactic compounds by which targets must be replaced.

The form of the *scope* and *target* fields is defined as follows, where $<\!f\!>$ stands for function names:

$$\begin{array}{lll} <\!target\!> & ::= & <\!f\!> \mid <\!f\!>.<\!loc\!> \mid <\!f\!>\texttt{@}<\!nat\!> \mid <\!expr\!> \\ <\!scope\!> & ::= & <\!f\!> \mid <\!f\!>.<\!loc\!> \\ <\!loc\!> & ::= & <\!nat\!> \mid <\!nat\!>.<\!loc\!> \end{array}$$

We illustrate the meaning of target and scope by the following examples:

- `f.2` denotes the second rewriting rule of the definition of the function `f`;
- `f.2.3` denotes the third condition of `f.2`. By convention, `f.2.0` stands for the conclusion of `f.2`;
- `f.2.3.1` denotes the left hand side of `f.2.3` and `f.2.3.2` stands for the right hand side of the same;
- `f@i` denotes the `i`th argument of `f`.

3.4 Synthesis of the offensive JCVM

Figure 4 contains the script used to generate from the defensive JCVM (which is over 10000 lines long) the offensive JCVM (5000 lines). The script begins with a preamble, which specifies the function to abstract, the name of the resulting function (at line 3) and the abstraction functions (at line 4). Then the remaining of the script is divided in two blocks of discarding commands (between lines 8 and 17) and substitution commands (between lines 21 and 26); there are no post-processing commands for this abstraction.

The block between lines 8 and 17 uses the discarding commands **`delete`** and **`drop`**.

- The command **`delete`** is used to suppress all occurrences of the target within the specified scope. For example at line 8, the script forces type verification operations (i.e. `type_verification`...) to be removed everywhere in the specification. By propagation, these commands may lead to rules being deleted, in particular because rules whose conclusion has been removed are also removed.
- The command **`drop`** is used to remove some arguments from the signature of a function. Such a command is needed to clean up code that is made vacuous by the abstraction. For example, the function `getfield_obj` takes as third argument some typing information which becomes vacuous in the offensive JCVM. Therefore we instruct the argument to be removed at line 17.

```
1  (* Preamble *)
2
3  build oexec from dexec using
4  abstraction functions alpha_do_prim alpha_do ...
5
6  (* Discarding commands *)
7
8  delete type_verification signature_verification Type_error
9         types_match test_exception_aastore assignment_compatible
10
11 delete test_exception_putfield_ref.2 test_exception_astore.3
12        test_exception_checkcast.4 test_exception_checkcast.5
13        test_exception_putstatic_ref.2
14
15 delete t0 in putstatic putfield_obj
16
17 drop getfield_obj@3 getstatic@1 tableswitch@1 lookupswitch@1
18
19 (* Substitution  commands *)
20
21 replace get_inst_from_heap by oget_inst_from_heap
22 replace v2t by (Ref Ref_null)
23 replace (trhi2vr (vr2tr vr) (absolu (v2z v))) by v in putfield_obj.36.0
24
25 select vp2z@1 v2z@1 vr2z@1 vr2hi@1 extr_from_opstack@2
26 select tpz2vp@2 in adr_aload res_opstack getfield_obj putfield_obj ...
```

Fig. 4. OFFENSIVE ABSTRACTION SCRIPT

The block between lines 21 and 26 uses the substitution commands **`replace`** and **`select`**.

- **`replace`** allows to replace the target by the argument specified in the **`by`** field. This argument can be a function name or an expression. This command allows the user to override the expression computed by the standard abstraction mechanism. We use this command at line 21 to provide a definition of the function `oget_inst_from_heap`, which takes as argument a heap `h` and a value `v`, and extracts from `h` the instance referenced by `v`. In this case, the purpose of the substitution is to provide a direct definition of the function; it is possible, but less convenient in this case, to use discarding commands for achieving the same definition. Observe that only three such substitutions are required to obtain a correct specification of the offensive JCVM.
- **`select`** allows to optimize the specification by projecting a function call onto one of its arguments. Such an optimization is useful when the abstraction produces a function `f` that behaves as a projection function, i.e. such that $\mathtt{f}(x_1, \ldots, x_n) = x_i$ for all $x_1 \ldots x_n$. An example of such function is given by `vp2z`. In the defensive JCVM, this function maps a typed value to its corresponding integer. In the offensive JCVM, the function behaves as the identity, so `(vp2z n)` is replaced by `n` as instructed at line 25. Note that the command affects the computational behavior of the functions, but not their meaning.

3.5 Synthesis of the abstract JCVM

```
 function alpha_da_prim : valu_prim → avalu_prim :=
⇒ alpha_da_prim (VReturnAddress v) → aReturnAddress v ;
⇒ alpha_da_prim (VBoolean v) → aBoolean ;
⇒ alpha_da_prim (VByte v) → aByte ;
⇒ alpha_da_prim (VShort v) → aShort ;
⇒ alpha_da_prim (VInt v) → aInt .

 function alpha_da : dstate → astate :=
⇒ alpha_da (Build_state sh hp s)
    → Build_astate (alpha_da_sheap sh) (alpha_da_stack s).
```

Fig. 5. Abstraction functions (excerpts)

Figure 5 contains some of the abstraction functions used to generate the abstract JCVM (which is around 5000 lines long) from the defensive JCVM. Observe how subtleties of the abstract JCVM are captured by the abstraction functions. For example:

- the abstract JCVM must keep track of return addresses to handle subroutines, and forget all other values: the definition of `alpha_da_prim` reflects this desiderata;
- abstract values do not carry any component that corresponds to the heap: this is reflected in the definition of `alpha_da`.

Figure 6 contains (excerpts of) the script used to generate the abstract JCVM (which is around 5000 lines long); the whole script is approximately 300 lines long. The preamble is immediately followed by a set of commands that act on the whole specification. These commands are used to discard operations and verifications involving the heap or values, for example `res_null`, `throwException` and `abortMemory` at line 10. Such commands are sufficient for handling various significant bytecodes, including `load`, `store`, `new`, `astore`, `goto`, `jsr` and `ret`.

The remaining of the script takes care of all bytecodes whose abstract semantics must be fine-tuned. This includes all bytecodes for method invokation and memory management, which are treated by discarding and substitution commands. We concentrate on branching bytecodes, as they are responsible for non-determinism and require the use of post-processing commands.

- The bytecodes `ifnull` and `ifnonnull` pop the top value of the operand stack and check whether it is null to determine the next program counter. The abstraction renders these functions non-deterministic. We handle this non-determinism with the command **`determine`** at line 18, whose effect is to transform the bytecodes into deterministic functions of target type `list astate`. The command, which implements the usual strategy of collecting the return values of a non-deterministic function into a list, also propagates the transformation to the function `aexec`.

```
1  (* Preamble *)
2
3  build aexec from dexec using
4  abstraction functions alpha_da_prim alpha_da ...
5
6  (* General commands *)
7
8  ...
9
10 delete throwException res_null  abortMemory
11
12 ...
13
14 (****** Commands for branching bytecodes ********)
15
16 ...
17
18 determine ifnull ifnonnull
19 drop res_scompare@1,2,3  res_acompare@1,2,3
20 determine res_scompare res_acompare
21 ...
```

Fig. 6. Typed abstraction script (excerpts)

- The bytecodes `if_scmp_cond`, `if_acmp_cond` and `if_cond` pop two values from the operand stack and compare them in order to determine the next program counter. This comparison is performed by the auxiliary functions `res_scompare` and `res_acompare`, but the abstraction renders these functions non-deterministic. A similar solution using **`determine`** applies, see line 20, but for technical reasons not discussed here, some arguments of these functions must be dropped before calling **`determine`**; the notation **`drop`** `f@1,2,3` at line 19 is a shorthand for **`drop`** `f@1 f@2 f@3`.

3.6 Automating cross-validation of virtual machines

In order to automate the proofs of cross-validation of the three JCVMs in Coq, we have designed specific reasoning tactics that generate induction principles for reasoning about complex recursive functions. These tactics yield drastic improvements; in particular, proof scripts are significantly easier to construct, and their size is reduced by 90% [5]. We expect that further automation is possible, and in particular that equational reasoning can be automated by adapting existing connections between Coq and external tools, such as Elan [1].

In a different line of work, we have been experimenting cross-validation of the JCVMs using Spike [12], an automatic theorem prover based on implicit induction. Currently Spike has been used to establish automatically the commutation properties for over two thirds of the instruction set, but we expect that further optimizations on Spike should allow us to handle even more instructions. This experience, which will be reported elsewhere, suggests that first-order theorem proving techniques could be advantageously exploited in the context of platform verification for smartcards.

3.7 Summary

Jakarta is at an early stage of development, but preliminary results support the claim laid in [6] that the construction and cross-validation of virtual machines could be automated to a large extent.

We have made good our promise to extract the offensive and abstract JCVMs from the defensive JCVM. One line of future research is to do likewise for novel type systems for Java(Card), see e.g. [4, 23, 31, 42]. More generally, we are interested in understanding whether our methodology applies to other typed low-level languages [2, 20, 47].

Another line of future research is to enhance Jakarta with new functionalities. Support for refinement is a requirement for incremental specification, and our most immediate goal. A longer-term goal is to device mechanisms that exploit the transformation logs produced by the JTK to guide the automatic construction of cross-validation proofs.

4 Conclusion

Formal verification of the JavaCard platform is the key to Common Criteria evaluations at the highest levels and a challenge for proof assistants. A recent survey [25] highlights the importance that formal methods have had in understanding and improving Sun's specifications of the Java(Card) platforms. But at the same time [25] observes a need to build appropriate environments that provide coherent support for formal specification, verification and provably correct implementations of the JavaCard platform. Such environments should incorporate general-purpose tools for developing and managing large-scale specifications and proofs, and specialized tools that are explicitly designed to reason about smartcard platforms. The design of such environments is a largely open avenue, and a promising field of research.

Acknowledgments We are especially grateful to Gustavo Betarte and Manuela Pereira for extensive comments on a draft of the paper.

References

1. C. Alvarado and Q.-H. Nguyen. ELAN for equational reasoning in COQ. In J. Despeyroux, editor, *Proceedings of LFM'00*, 2000. Rapport Technique INRIA.
2. D. Aspinall and A. Compagnoni. Heap-bound assembly language. Manuscript, 2001.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02.* IEEE Computer Society Press, 2002.
5. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In C. Muñoz and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. To appear.

6. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.
7. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002.
8. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
9. J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.
10. M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.
11. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic Purse Applet Certification. In S. Schneider and P. Ryan, editors, *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2000.
12. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, January 1997.
13. L. Casset. Development of an Embedded Verifier for JavaCard ByteCode using Formal Methods. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of FME'02*, Lecture Notes in Computer Science, 2002. To appear.
14. L. Casset, L. Burdy, and A. Requet. Formal Development of an Embedded Verifier for JavaCard ByteCode. In *Proceedings of DSN'02*. IEEE Computer Society, 2002.
15. L. Casset and J.-L. Lanet. A Formal Specification of the Java Byte Code Semantics using the B Method. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proceedings of Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999.
16. Z. Chen. *Java Card for Smart Cards: Architecture and Programmer's Guide*. The Java Series. O'Reilly, 2000.
17. Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP'02*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, 2002.
18. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.2*, January 2002.
19. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'00*, pages 439–448. ACM Press, 2000.
20. K. Crary and G. Morrisett. Type structure for low-level programming languages. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, 1999.
21. Common Criteria. `http://www.commoncriteria.org`.
22. e-Europe SmartCards. See `http://eeurope-smartcards.org`.
23. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, November 1999.

24. E. Giménez and O. Ly. Formal modeling and verification of the java card security architecture: from static checkings to embedded applet execution, 2002. Talk delivered at the Verificard'02 meeting, Marseille, 7-9 January 2002. Slides available at `http://www-sop.inria.fr/lemme/verificard/2002/programme.html`.
25. P. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.
26. JavaCard Technology. `http://java.sun.com/products/javacard`.
27. JML Specification Language. `http://www.jmlspecs.org`.
28. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. Submitted.
29. Gemplus Research Labs. Java Card Common Criteria Certification Using Coq. Technical Report, 2001.
30. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
31. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 200x. To appear.
32. K. R. M. Leino. Extended Static Checking: A Ten-Year Perspective. In R. Wilhelm, editor, *Informatics–10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175, 2001.
33. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
34. X. Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Proceedings e-Smart 2001*, volume 2140, pages 150–164. Springer-Verlag, 2001.
35. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, 2000.
36. G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
37. T. Nipkow. Verified Bytecode Verifiers. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001.
38. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
39. J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Proceedings of the ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.
40. A. Requet. A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of FMICS'00*, pages 29–46, 2000.
41. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
42. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.
43. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Platform Security*, 2001. Technical White Paper.

44. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Application Programming Interface (API)*, 2002.
45. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Runtime Environment (JCRE) Specification*, 2002.
46. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Virtual Machine Specification*, 2002.
47. K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Manuscript, 2001.