A Tool-Assisted Framework for Certified Bytecode Verification

Gilles Barthe and Guillaume Dufay

INRIA Sophia-Antipolis, France
{Gilles.Barthe,Guillaume.Dufay}@sophia.inria.fr

Abstract. Bytecode verification is a key security function in several architectures for mobile and embedded code, including Java, JavaCard, and .NET. Over the last few years, its formal correctness has been studied extensively by academia and industry, using general purpose theorem provers. Yet a recent roadmap on smartcard research [1], and a recent survey of the field of Java verification [11], point to a severe lack of methodologies, techniques and tools to help such formal endeavours. In earlier work, we have developed, and partly automated, a methodology to establish the correctness of static analyses similar to bytecode verification. The purpose of this paper is to complete the automation process by certifying the different dataflow analyses involved in bytecode verification, using the Coq proof assistant. It enables us to derive automatically, from a reference virtual machine that performs verification at run-time, and satisfies minimal requirements, a provably correct bytecode verifier.

1 Introduction

Several architectures for mobile and embedded code, including Java, JavaCard, and .NET, feature a bytecode verifier (BCV), which performs a modular (i.e. method per method) static analysis on compiled programs prior to their loading, and rejects potentially insecure programs that may violate type safety, or perform illegal memory accesses, or not respect the initialization protocol, or yield stack underflows or overflows, *etc.*

The bytecode verifier is a key security function in these architectures, and as such its design and implementation must be correct. Over the last few years, a number of projects have been successful in proving formally that bytecode verification is correct, in the sense that it obeys the specification of Sun. Many of these projects were carried by smartcard industrials in the context of security evaluations; for example, Schlumberger Cards and Terminals was recently awarded an EAL7 Common Criteria[1] certificate for their formal models of the JavaCard platform.

[1] The Common Criteria is an international evaluation scheme for the security of IT products. It features seven evaluation assurance levels (EAL). The highest quality levels EAL5 to EAL7 impose the use of formal methods for the modelling, specification and verification of the product being certified.

Yet such projects are labour-intensive, hence costly, and can only be conducted by a few experts in formal methods. A recent roadmap on smartcard research [1], and a recent survey of the field of Java verification [11], point to a severe lack of methodologies, techniques and tools to conduct cost-effective security evaluations.

Our work aims at developing tools and libraries that help validate execution platforms for mobile and embedded code. The research reported here focuses on specifying and proving the correctness of bytecode verification, following a methodology that is common to many existing works in the area, see e.g. [2, 4, 5, 16]. The methodology factors the effort into two phases:

– *virtual machines specification and cross-validation (VM phase):* during this first phase, one provides the specification of several virtual machines, including a defensive virtual machine that manipulates typed values and performs type-checking at run-time (as well as computations), and abstract virtual machine that only manipulates types and only performs type-checking. One also cross-validates these different virtual machines, e.g. one shows that the abstract virtual machine detects all typing errors that may occur during the execution of a program on the defensive virtual machine;
– *bytecode verification specification and verification (BV phase):* during this second phase, one builds and validates the bytecode verifier, using the abstract virtual machine defined during the VM phase. First, it involves modeling a dataflow analysis – for an unspecified execution function that meets minimal requirements. Second, it involves proving the correctness of the analysis; essentially it amounts to showing that the analysis will reject all programs that go wrong during execution. Third, it involves instantiating the analysis to the virtual machine defined in the VM phase, using cross-machine validation to establish that the VM enjoys all properties assumed for the unspecified execution function in the definition of the dataflow analyses; the instantiation provides a bytecode verifier, and a proof of its correctness. Note that different algorithms may be chosen, so as to account for some advanced features, e.g. subroutines and initialization in the Java and JavaCard platforms, or to minimize resource usage during verification, see Section 2.

In earlier work, we have been developing Jakarta, an environment which supports the specification and cross-validation of the virtual machines [3], and offers a high level of automation for performing the VM phase. In a nutshell, Jakarta consists of a specification language JSL in which virtual machines can be described, an abstraction engine that transforms virtual machines (e.g. that extracts an abstract virtual machine from a defensive one), and an interface with theorem provers, which maps JSL specifications to the prover specification language and generates automatically correctness proofs for the cross-validation of virtual machines. We mostly use Jakarta in conjunction with the proof assistant Coq [8], although prototypes interfaces to Isabelle [18] and PVS [20] exist[2].

[2] One particular reason for our choice is that French evaluation bodies recommend the use of Coq or B to carry Common Criteria evaluations at the highest levels.

The purpose of this paper is to complement our earlier work by providing a modular framework for performing the BV phase. Starting from an abstract notion of virtual machine on which we only impose minimal assumptions, we build a parametric bytecode verifier that encompasses a large class of algorithms for bytecode verification, and show the algorithm to be correct in the sense that it will reject programs that may go wrong. One novelty of our framework is to provide a high-level proof that it is sound to perform bytecode verification on a method per method basis. Another novelty is to provide a generic bytecode verifier that can be instantiated to several analysis including standard analyses that only accept programs with monomorphic subroutines, set-based analyses that accept programs with polymorphic subroutines, as well as other analyses for which no formal correctness proof was previously known. From a more global perspective, the combination of our framework for bytecode verification with the Jakarta toolset yields an automated procedure to derive a certified bytecode verifier from a reference defensive virtual machine; the procedure is applicable to many settings, and has been successfully used to certify the JavaCard platform. We return to these points in the conclusion.

Contents of the Paper. The remaining of the paper is organized as follows. We begin in Section 2 with a brief introduction to Coq and its module system, and with a brief overview of bytecode verification. We proceed in Section 3 with the basic definitions and constructions underlying bytecode verification. Section 4 and Section 5 are respectively devoted to formalizing and certifying a parameterized verification algorithm and compositional techniques that justify the method-per-method verification suggested by Sun. In Section 6, we show how the framework may be instantiated to different analyses. We conclude in Section 7 with related work, a general perspective on our results thus far, and directions for further research.

2 Preliminaries

2.1 Principles and Algorithms of Bytecode Verification

Bytecode verification [9, 17] is a static analysis that is performed method per method on compiled programs prior to their loading. Its aim is to reject programs that violate type safety, perform illegal memory accesses, do not respect the initialization protocol, yield stack underflows or overflows, *etc.*

The most common implementation of bytecode verification is through a dataflow analysis [13] instantiated to the abstract virtual machine that operates at the type level. The underlying algorithm relies on a history structure, storing the computed abstract states for each program point, and on an unification function on states. Then, starting from the initial state for a method, it computes a fixpoint with the abstract execution function. If the error state does not belong to the resulting history structure then bytecode verification is successful.

In the standard algorithm, called monovariant analysis, the history structure only stores one state for each program point and the unification function unifies, performing a join on the JavaCard type lattice, the computed state (resulting from one step of abstract execution) and the stored state. Unfortunately, this algorithm does not accept polymorphic subroutines (subroutines called from different program points). To handle such subroutines, the history structure must contain a set of states for each program point. For the polyvariant analysis, the unification function adds the computed state to the corresponding set from the history structure. This technique needs much more memory than monovariant analysis, however, it is possible to perform state unification rather than set addition in most cases. This last technique, called hybrid analysis (as described in [7, 12]), offers the best compromise between memory consumption, precision and efficiency.

Our framework also deals with lightweight bytecode verification [19], a special kind of verification that can fit and run in chips used for smart cards, but due to space constraints details are omitted.

2.2 The Coq Proof Assistant

Coq [8] is a general purpose proof assistant which is based on the Calculus of Inductive Constructions, and which features a very rich specification language and a higher-order predicate logic. However, we only use neutral fragments of the specification language and the logic, i.e. fragments which are common to several proof assistants, including Isabelle and PVS. More precisely, we use first-order logic with equality, first-order data types, structural recursive definitions, record types, but no dependent types – except in the definition of `gfp`, but such a function is also definable in Isabelle and PVS. Furthermore, Coq underlying logic is intuitionistic, hence types need not have a decidable equality. For the sake of readability, and because it is specific to Coq, we gloss over this issue in our presentation[3].

Modules. Our framework makes an extensive use of the interactive ML-style modules that were recently integrated to Coq [6]. Hence we briefly review the syntax for modules. The keyword **`Module Type`** introduces the declaration of a type of a module, and is followed by its name, a collection a **`Parameter`** and **`Axiom`** declarations giving its signature, and it is closed by the keyword **`End`**. A module type can also include (and, in a certain sense, extend) other module types with the keyword **`Declare Module`**. A module type is implemented using the keyword **`Module`** (the module type it satisfies is specified after the notation `<:`). As usual, the module must fulfill the signature of the module type it implements. Note that other modules can be given as parameters of a module. Finally, constructions of a module can be accessed outside the module using the dot notation of qualified names or directly with the keyword **`Import`** followed by the module name.

[3] Although our framework addresses decidability by making appropriate assumptions in modules, we omit such assumptions in this paper.

Notations. The type of propositions is `Prop`, and the type of data is `Set`. The types `predicate A` and `relation A` respectively denote the set of predicates and binary relations over a type `A`.

We conclude with some basic definitions used throughout the paper. Given `A : Set`, $<_A$`: (relation A)`, `f : A→A` and `P : (predicate A)`, we let $\leq_A$ denote the reflexive closure of $<_A$ and define

```
  (monotone <_A f) ≡ ∀a,a':A.(a <_A a') →((f a) ≤_A (f a'))
 (decreases <_A f) ≡ ∀a:A.((f a) ≤_A a)
(down_closed <_A P) ≡ ∀a,a':A.(a <_A a')→(P a')→(P a)
```

Finally we let `(well_founded` $<_A$`)` state that the relation $<_A$ is well founded, i.e. that there is no infinite decreasing chain.

3 Bytecode Verification as a Fixpoint Computation

We favour a definition that abstracts away from implementation details, and define a bytecode verifier as a predicate that rejects programs that may go wrong.

Definition 1.

– *A* transition system with error *(TSE) is given by a type* `state` *of* states*, an* execution relation `exec` *over states, and a set* `err` *of error states. Formally,*

```
Module Type TSE.
Parameter state : Set.
Parameter exec  : (relation state).
Parameter err   : (predicate state).
End TSE.
```

We say that a state `a` *of a given TSE is bad, written* `bad a`*, if it can reach an error state by successive transitions of the execution relation.*

– *A* bytecode verifier *over a module* `tse` *of type* `TSE` *is given by a predicate* `check` *that rejects all bad states. Formally, the module* `BCV` *of bytecode verifiers extends the module* `TSE` *as follows*[4]*:*

```
Module Type BCV.
Declare Module tse: TSE. Import tse.
Parameter check : (predicate state).
Axiom ∀ a:state.(check a) →¬(bad a).
End BCV.
```

The standard way to build a bytecode verifier is to endorse the type of states with a well-founded order for which execution is decreasing (to guarantee termination), and such that error states are downwards closed. If furthermore execution is deterministic, one can compute for every state `a`, the greatest fixpoint `b` below `a`; then it is sufficient to check that `b` is not an error state to conclude that `a` is not bad.

[4] Coq modules provide names for axioms, so that these axioms can later be used in proofs. For readability we omit names of axioms in our module declarations.

Definition 2. *A* fixpoint structure with errors *(FSE) is given by the module*

```
Module Type FSE.
Parameter state  : Set.
Parameter exec   : state → state.
Parameter err    : (predicate state).
Parameter <state : (relation state).

Axiom (well_founded <state).
Axiom (decreases <state exec).
Axiom (monotone <state exec).
Axiom (down_closed <state err).
End FSE.
```

We can define a module functor satisfying, from a module of type FSE, the type of the module BCV:

```
Module FSE2BCV [fse:FSE] <: BCV.
```

To do so, we first define for every state `a` of a FSE the greatest fixpoint `gfp a` below it as

$$\texttt{gfp a} = \begin{cases} \texttt{a} & \text{if exec } \texttt{a = a} \\ \texttt{gfp (exec a)} & \text{otherwise} \end{cases}$$

Then, we define `check a` as `¬(err_state (gfp a))`. As execution is monotone and `gfp a` is the greatest fixpoint below `a`, it is clear that such a checking is sufficient to guarantee that `a` is not a bad state.

4 A Parameterized Bytecode Verifier

In this section, we construct a parameterized bytecode verifier that rejects programs that may go wrong when executed with an abstract virtual machine. We start with the definition of the latter.

Definition 3. *An abstract virtual machine (AVM) is given by an ordered type of states* `state` *endorsed with a downwards closed set of errors* `err`*, a type of locations* `loc`*, an execution function* `exec`*, a successor function* `succs` *that computes the successors of a state and an enumeration* `locs` *of the locations of the program. Formally,*

```
Module Type AVM.
Parameter state : Set.
Parameter <state : (relation state).
Parameter err   : (predicate state).
Parameter loc   : Set.
Parameter succs : loc → state → (list loc).
Parameter locs  : (list loc).
Parameter exec  : loc → state → state.

Axiom (down_closed <state err).
End AVM.
```

Bytecode verification relies on stackmaps, i.e. functions that associate to every program point a history structure. History structures can be seen as an abstraction of the mathematical set notion.

Definition 4. *The module type* `History_Struct` *of* history structures *is parameterized by a carrier set* `A`[5] *and given a type constructor* `hist`*, a projector function* `hist_unit`*, an extension* `hist_less` *of an order on* `A`*, an decreasing iterator* `hist_foldr` *and a membership predicate* $\in_{\mathtt{hist}}$ *on which we define an existential predicate* $\exists_{\mathtt{hist}}$*. Formally,*

```
Module Type History_Struct [A:Set].
Parameter hist      : Set → Set.
Parameter hist_unit : A → (hist A).
Parameter hist_less : ∀ <_A:(relation A). (relation (hist A)).
Parameter ∈_hist      : A →(hist A) →Prop.
Axiom ∀ x:A.(x ∈_hist (hist_unit x)).

Parameter hist_foldr : ∀ B:Set.(A→B→B) → B→(hist A)→ B.
Axiom ∀ B:Set. ∀ f:(A →B →B). ∀ <_B:(relation B).
 (∀ a:A.(decreases <_B (f a))) →
 (∀ a:(hist A).(decreases <_B (λ b:B.(hist_foldr B f b a)))).

Definition ∃_hist := λ P:(predicate A) λ s:(hist A)
  ∃ a.(P a) ∧ (a ∈_hist s).
Axiom ∀ <_A:(relation A). ∀ P:(predicate A).
  (down_closed <_A P) →(down_closed (hist_less <_A) (∃_hist P)).
End History_Struct.
```

In the following, we will use the notation $<_{\mathtt{A}}^{\mathtt{hist}}$ for the extension `(hist_less <_A)` of a relation $<_{\mathtt{A}}$.

The definition of stackmaps is included in a module of stackmap structures. Essentially, a stackmap structure consists of an abstract virtual machine, of a history structure, and of a unification function that merges states and that is decreasing and monotone w.r.t. the order inherited from the history structure. In order to construct a fixpoint structure from a stackmap structure, we also require histories $<_{\mathtt{A}}^{\mathtt{hist}}$ to be well-founded and a supremum for $<_{\mathtt{A}}$.

Definition 5. *The module SMS of* stackmap structures *is defined as:*

```
Module Type SMS.
Declare Module avm : Abstract_VM. Import avm.
Declare Module hs  : (History_Struct state). Import hs.

Parameter unify : state → (hist state) → (hist state).
Axiom ∀ s:state.(decreases <^hist_state (unify s)).
Axiom ∀ s:state.(monotone <^hist_state (unify s)).
Axiom ∀ s:state.∀ s':(hist state).(unify s s')=s' →
∃ y.(y ≤_state s) ∧ (y ∈_hist s').
```

[5] In reality, Coq modules type can only be parameterized by other modules, so one has to use a module that is “isomorphic” to `Set`.

```
Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤state ⊤).

Axiom (well_founded <hist state).
End SMS.
```

One can define the type `stackmap` of stackmaps over a stackmap structure `sms` as `list (sms.avm.loc * (sms.hs.hist sms.avm.state))`, and an execution function over stackmaps `sms_exec: stackmap →stackmap` which corresponds to the recursive procedure in Kildall's algorithm [13]. It is straightforward to derive a fixpoint structure, and hence a bytecode verifier, for the TSE induced by `sms_exec: stackmap →stackmap`. In order to conclude that the resulting fixpoint structure also yields a bytecode verifier for the TSE induced by the AVM, one needs to observe that the following diagram commutes:

$$\begin{tikzcd}
\texttt{avm.loc*avm.state} \arrow[r, "\texttt{avm.exec*avm.next}"] \arrow[d, "\texttt{make\_stackmap}"] & \texttt{avm.loc*avm.state} \arrow[d, "\le\texttt{make\_stackmap}"] \\
\texttt{stackmap} \arrow[r, "\texttt{sms\_exec}"] & \texttt{stackmap}
\end{tikzcd}$$

Here `make_stackmap` denotes the function that takes as input a pair ⟨`l`, `a`⟩, and returns as output the stackmap in which the program point `l` is associated to the singleton history `hist_unit a`, and every other program point is associated to ⊤.

5 Correctness of Bytecode Verification

In the previous section, we have shown that programs that pass bytecode verification do not go wrong when executed on an abstract virtual machine which satisfies minimal requirements. The purpose of this section is to lift this result to a defensive virtual machine: more precisely, we are going to show that programs that pass bytecode verification do not go wrong when executed on a defensive virtual machine which satisfies minimal requirements. It involves relating a defensive and an abstract virtual machine, and proving that no difficulty arises through exception handling (which is performed by the defensive virtual machine, but ignored by the abstract one), or through method invokation (which remains within the same frame for the abstract virtual machine, as explained below). We begin by defining defensive virtual machines.

Definition 6. *A* defensive virtual machine *(DVM) is given by a type* `state` *of states, an execution function* `exec`*, a type* `frame` *of frames, an accessor function* `getstack` *that associates to each state a list of frames (i.e. its stack), another accessor function* `getinstr` *that associates to each state the nature of the next execution to be executed, and a set* `err_frame` *of error frames. Formally,*

```
Module Type DVM.
Parameter state     : Set.
Parameter exec      : state → state.
Parameter frame     : Set.
Parameter getstack  : state → (list frame).
Parameter getinstr  : state → type_of_instr.
Parameter err_frame : (predicate frame).
End DVM.
```

The function `getinstr` distinguishes between 4 cases: execution is intra-procedural `sameframe` (that acts only in the current frame, e.g. for arithmetic instruction, branching instruction, *etc*), execution is a method invokation `invoke`; execution is a return step `return` (pops a frame); or execution raises an exception `exception`.

We now turn to formulating a set of general properties about method invokation and exception handling, and proving that such properties ensure that programs that pass bytecode verification will not go wrong. These properties involve an abstract virtual machine and an abstraction function.

Abstract Virtual Machine. We assume given an abstract virtual machine `avm`, with a function `init` that returns for each method or exception the corresponding initial state. Furthermore, we assume given a decomposition of abstract method invokation in two functions, so as to be able to simulate the modifications made by the concrete virtual machine on a frame when the control flow is given to the invoked method and when it returns to the invoker method. Formally, we assume given two functions `exec_invk` and `exec_ret` whose composition is equal to `avm.exec` for states `a` such that `getinstr a = invoke`.

Abstraction Function. We assume given a function that maps a frame to an abstract state and a location α: `dvm.frame` →`avm.state`. The function is extended a function β: `dvm.state` →`avm.state`. on defensive states by abstracting the topmost frame of the stack (if the stack is empty, we return a default error value).

Safe States. We now turn to the definition of safe abstract states. A abstract state will be safe if it is greater than a state belonging to the history structure computed by the abstract bytecode verifier at the location of the given state. This notion is extended to defensive frames by abstraction.

The notion of safety for a defensive state must guarantee that the stack is well-formed, i.e. that all the frames below the top one are in an "intermediate" state which is not reached by the abstract virtual machine until the invoked method returns. Then, a defensive state `s` will be safe if lstinline!getstack s = []
! or if `getstack a = f::lf` and each frame in `lf` is of the form `exec_invk f'` where `f'` is a safe frame.

We now show that safe states are closed under execution and are not bad.

Lemma 1. *Let* `s` *be a defensive state. Suppose:*

- *if* `getinstr s = sameframe`*, then the following property holds:*

$$\texttt{(avm.exec (}\beta\texttt{ s))} \leq_{\textsf{state}} \texttt{(}\beta\texttt{ (dvm.exec s))}$$

- *if* `getinstr s = invoke` *and* `getstack s = f::lf`*, then the following properties hold:*

$$\exists\ \texttt{f':frame. getstack (dvm.exec f') = f'::(exec\_add f)::lf}$$
$$\texttt{(init (}\beta\texttt{ s))} \leq_{\textsf{state}} \texttt{(}\beta\texttt{ s)}$$

- *if* `getinstr s = return` *and* `getstack (dvm.exec s) = f::lf`*, then there exists two frames* `f'` *and* `f''` *such that the following properties hold:*

$$\texttt{getstack s = f'::f''::lf}$$
$$\texttt{(aexec\_ret (}\alpha\texttt{ f''))} \leq_{\textsf{state}} \texttt{(}\alpha\texttt{ f)}$$

- *if* `getinstr s = exception` *and* `getstack s = f::(lf@lf')`*, then the following properties hold:*

$$\exists\ \texttt{f':frame. getstack s = f'::lf'}$$
$$\texttt{(init (}\beta\texttt{ s))} \leq_{\textsf{state}} \texttt{(}\beta\texttt{ s)}$$

If furthermore `s` *is safe, then* `dvm.exec s` *is also safe.*

This lemma if proved using properties on the bytecode verifier and property on `exec_invk` and `exec_ret` w.r.t. `avm.exec`. Then one can easily construct a bytecode verifier for a defensive virtual machine `dvm`. Formally, from a module type `Comp_Struct`, that contains all the assumptions of Lemma 1, we are able to define a functor module `BCV_dexec` satisfying the module type `BCV` for the execution `dvm.exec`. The function `check` of the module type `BCV` is defined assuming that the given defensive state is safe and that the result of bytecode verification for all initial states (methods and exceptions) of the program does not contain an error state. Finally, by Lemma 1, we can prove the property `check_ok` of the module type `BCV`, stating that if the verification `check` was successful, we can not reach with the defensive virtual machine an error state.

6 Instantiation of History Structures

The previous section describes the construction of a correct bytecode verifier for a defensive virtual machine. The construction is parameterized by a structure that records the history of the computations performed by the verifier. The purpose of this section is to present different instantiations of our framework, focusing on different choices of history structures that correspond to the algorithms described in Section 2. We use these instantiations as convenient entry points in our formalization, see Subsection 7.2.

Monovariant Analysis. A *monovariant analysis* (MA) is given by a well-founded order on states with a supremum, and by proofs that the execution function is monotone w.r.t. the order on states and the unification is decreasing and monotone. Formally,

```
Module Type Monovariant_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter unify  : state → state → state.

Axiom ∀ s:state.(decreases <state (unify s)).
Axiom ∀ s:state.(monotone <state (unify s)).
Axiom ∀ s,s':state.(unify s s')=s' →∃ y.(y ≤state s) ∧ (y ∈hist s').

Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤state ⊤).

Axiom (well_founded <state).
Axiom ∀ l:loc.(monotone <state (exec l)).
End Monovariant_Analysis_Struct.
```

The construction of a stackmap structure from a monovariant analysis is done by a functor module `Monovariant_Analysis` from the previous module type definition. It mainly proceeds by instantiating the parametric history structure to the identity history structure, in which `hist A` is defined as `A`, and the other fields are instantiated in the obvious way.

Polyvariant Analysis. A *polyvariant analysis* is given by a natural number `max_length_set` that fixes the maximal size of the set of abstract states associated to each program point, by a supremum state ⊤, by an error state `err_st` and by a proof that execution is monotone. Formally,

```
Module Type Polyvariant_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤state ⊤).

Axiom ∀ l:loc.(monotone <state (exec l)).
End Polyvariant_Analysis_Struct.
```

One proceeds by instantiating the history structure in such a way that `hist A` is defined as the set of elements of `A` of cardinal less than `max_length_set` (the other fields are instantiated in the obvious way). Then, this module is used with the functor `Polyvariant_Analysis` to construct a stackmap structure, defining the function `unify` as:

```
λ a:state λ s:(hist state)
(if ((set_size (set_add a s)) < max_length_set)
 then (set_add a s)
 else (set_add err_st s))
```

In that case, `hist_less` does not use the order $<_{\mathtt{state}}$ and is defined as set inclusion. It is interesting to notice that the polyvariant analysis is by far the simplest algorithm to instantiate.

Hybrid Analysis. An *hybrid analysis* is given combining elements needed by monovariant and hybrid analysis and adding an optimization function `opt_unify` to discriminate in which cases the unification of states must take place. Formally,

```
Module Type Hybrid_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter opt_unify : state → state → bool.
Parameter unify  : state → state → state.

Axiom ∀ s:state.(decreases $<_{state}$ (unify s)).
Axiom ∀ s:state.(monotone $<_{state}$ (unify s)).
Axiom ∀ s,s':state.(unify s s')=s' →∃ y.(y $\leq_{state}$ s) ∧ (y $\in_{hist}$ s').

Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter ⊤ : state.
Axiom ∀ a:state.(a $\leq_{state}$ ⊤).

Axiom (well_founded $<_{state}$).
Axiom ∀ l:loc.(monotone $<_{state}$ (exec l)).
End Hybrid_Analysis_Struct.
```

The same history structure as polyvariant analysis is used for the hybrid analysis. The function `unify` is then defined as follow :

```
λ a:state λ s:(hist state)
(Case (set_map_bool opt_unify unify a s) of
 (Some res) ⇒ res |
  None ⇒ (if ((set_size (set_add a s)) < max_length_set)
           then (set_add a s)
           else (set_add err_st s))
 end)
```

where `set_map_bool` ranges over elements of `s`, performs unification depending on the result of `opt_unify` and returns the resulting set if unification has occurred or `None` otherwise. In that case, `hist_less` combines the order $<_{\mathtt{state}}$ on states and set inclusion.

7 Conclusions

7.1 Related Work

As mentioned in the introduction, there is a considerable body of machine-checked specifications of execution platforms such as the JVM or .NET, many of which use the methodology instrumented in our work. A notable exception is the extensive account of bytecode verification developed by Klein, Nipkow, and Wildermoser [14, 15] using the proof assistant Isabelle [18]. For lack of space, we refer the reader to [11, 17] for a more comprehensive account of related work.

There are also machine-checked proofs of type soundness for .NET [10, 21]. This work is more closely related to ours in the sense that [21] explicitly aims at developing tools to automate type soundness proofs. The major difference with our work is that they do not pursue cross-machine validation, and opt instead for a standard type soundness proof.

7.2 Perspectives

We have develop a general framework that establishes the correctness of a parameterized bytecode verifier, and justifies the compositional techniques of bytecode verification. The framework has been instantiated for specific history structures that are often considered in the literature and implementations. These instantiations provide convenient entry points to our framework, and can be used in combination with Jakarta to build and validate bytecode verifiers with a high degree of automation. As illustrated in Figure 1, such a combination requires the user to provide:

– a defensive virtual machine;
– the definition of abstraction functions, in the form of Jakarta abstraction scripts, that are used to construct the abstract virtual machine and an offensive virtual machine[6]. Scripts may contain some minimal amount of proof information to carry cross-machine validation;
– a formal proof of the correctness w.r.t. bytecode verification of method invokation and exception handling, i.e. an instantiation of the module `Struct_Comp` of Section 5;
– an instantiation of the history modules to the abstract virtual machine generated by Jakarta;

and returns an offensive virtual machine, several bytecode verifiers, and a proof that these bytecode verifiers are correct, in the sense that they will reject programs that go wrong on the defensive virtual machine, and that the offensive and defensive virtual machines coincide on programs that are accepted by bytecode verification.

Such a combination has been used to good purpose for validating the JavaCard platform. Using Jakarta, we have generated from a defensive virtual machine

[6] Such a machine manipulates untyped values, and relies on the bytecode verifier to detect programs that may go wrong.

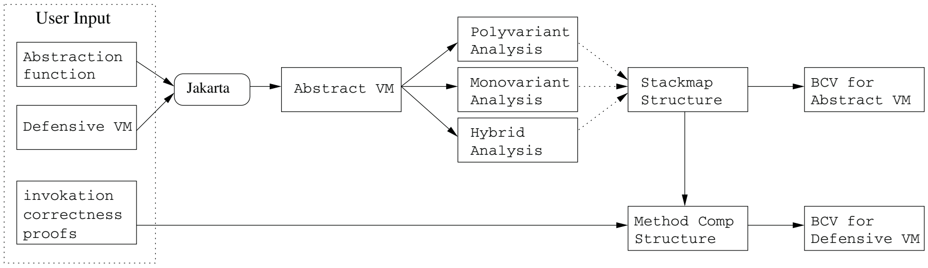

Fig. 1. Framework architecture

(10,000 lines of code), both an abstract and an offensive virtual machine (5,000 lines of code each), as well as more than 10,000 lines of proof scripts that establish cross-machine validation and the monotonicity of the generated abstract virtual machine. We have provided another 1,500 lines of proof scripts which concern the correctness w.r.t. bytecode verification of method invokation and exception handling. Together with the output of Jakarta, these 1,500 lines provide all relevant information for the bytecode verifier to be proved correct – without any further user interaction.

As to future work, we plan to instantiate our framework to enhanced bytecode verifiers that guarantee a stronger security of applications. Indeed, there have been many proposals of type systems for the JVM that provide stronger guarantees with respect to safety and security, and it would be interesting to adapt our virtual machine specifications to such type systems, and use the framework described here to derive certified bytecode verifiers based on these type systems. In fact, we have started modeling a defensive JVM machine for an information flow type system[7], and intend to use the framework described in this paper, in combination with Jakarta, to build and validate a bytecode verifier for information flow. Likewise, it would be interesting to apply our methodology to other execution platforms, such as the .NET platform.

References

1. Roadmap for European Research on Smartcard Technologies. See `http://www.ercim.org/reset`
2. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 335 – 351. Springer-Verlag, 2003.

[7] Non-interference is a property about all program executions and thus it cannot be completely enforced by a defensive virtual machine. However, we only use the defensive virtual machine as a tool to prove that the bytecode verifier enforces non-interference, without making any claim on the security guarantees provided by such a defensive virtual machine.

3. G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In H. Kirchner and C. Ringessein, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.
4. 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.
5. G. Betarte, B. Chetali, E. Giménez, C. Loiseaux, and O. Ly. Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution. In *Proceedings of ESMART'02*, 2002.
6. J. Chrzaszcz. Implementing Modules in the Coq System. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 270 – 286. Springer-Verlag, 2003.
7. A. Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *Proceedings of FTFJP'02*, 2002.
8. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.4*, February 2003.
9. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
10. A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of POPL'01*, pages 248–260. ACM Press, 2001.
11. 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.
12. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
13. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.
14. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
15. G. Klein and M. Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, December 2003.
16. 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.
17. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
18. 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.
19. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, October 1998.
20. 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.
21. D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In M. Baaz and A. Voronkov, editors, *Proceedings of LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 418–434, 2002.