Under consideration for publication in Formal Aspects of Computing

Formalizing Non-Interference for a Simple Bytecode Language in Coq

Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. In this paper, we describe the application of the interactive theorem prover Coq to the security analysis of bytecode as used in Java. We provide a generic specification and proof of non-interference for bytecode languages using the Coq module system. We illustrate the use of this formalization by applying it to a small subset of Java bytecode. The emphasis of the paper is on modularity of a language formalization and its analysis in a machine proof.

Keywords: Formal Methods for Security, Programming Language Analysis, Modular Specification, Interactive Theorem Proving

1. Introduction

An important application domain of interactive theorem proving, and formal techniques in general these days, is the analysis of programming languages and their related compilers and run-time systems, most prominently the Java language and the related Java Virtual Machine (JVM) infrastructure. Java is a suitable target for interactive theorem proving techniques as it is ubiquitous and therefore often applied in security and safety-critical domains. Java is also a suitable target for formal analysis because its definition is relatively clean. Hence, the high effort naturally coming along with a mechanical formal analysis is justified. In this paper, we report on our work of formalizing and analyzing confidentiality properties of the Java bytecode language with the Coq [Coq04] theorem prover. The actual confidentiality property we prove is non-interference, a property that excludes the illegal flow of information from any confidential parts to public parts of programs, thereby providing confidentiality.

Apart from just proving a property, this rather big case study provides the implementation of a certified bytecode verifier for non-interference: as the Coq proof assistant is based on constructive logic, we can extract an executable program from our mechanized proof.

In the remainder of this introductory section, we introduce theorem proving in Coq. In Section 2, we give an account on the type safety and security verification of Java and the JVM prior to our work and explain non-interference. Then, in Section 3, we introduce our formalization of a Coq framework for non-interference

Correspondence and offprint requests to: Florian Kammüller, TU Berlin, FR 5-6, Franklinstr. 28/29, 10587 Berlin, Germany. e-mail: flokam@cs.tu-berlin.de

of bytecode that enables the extraction of a certified bytecode verifier [BK05]. The mechanization restrains itself to a simple imperative subset of Java bytecode, but using Coq's module concept it represents a generic framework parameterizing the programming language and its operational semantics. The emphasis of the exposition is on the framework, i.e. the proof of non-interference, the modular abstractions and the extraction of a bytecode verifier rather than the extension to the full JVM bytecode.

Coq

Interactive theorem proving is more and more successfully being applied in programming language analysis. This is due to the fact that interactive provers implement expressive logics, usually some kind of Higher Order Logic (HOL) [Chu40]. The interactive theorem prover Coq is also a HOL system, differing, however, in quite a few respects from classical HOL systems. The main difference from a users perspective is that the logic in Coq is not classical. The basis for Coq is a constructive type theory that is interpreted as a logic according to the Curry-Howard isomorphism [How80]. In this isomorphism, types are interpreted as propositions and terms inhabiting types as proofs of those propositions. For example, a proof of an implication $A \Rightarrow B$ is a function transforming an element of type A, i.e. a proof of proposition A, into an element of type B, i.e. a proof of proposition B. Coq represents a constructive logic. Therefore, some classical axioms like, for example, the classical axiom called *tertium-non-datur* $A \vee \neg A$ (*tnd*) are not represented in Coq. That is, if we would like to prove an implication, like $A \Rightarrow B$, we cannot – as is a common way of proving implications – assume $A \wedge \neg B$ and then show a contradiction. This reasoning is based on the classical view that the implication $A \Rightarrow B$ corresponds to $\neg A \vee B$ and the assumption of the axiom *tnd*.[1] Instead, to prove an implication, we have to construct a function as described above. This is just an example to illustrate the basic ideas of constructive logic. For a comprehensive introduction to type theory and constructive logics in its various forms, see [Tho90].

Although constructivity imposes restrictions that can be quite awkward at times, it offers one decisive advantage: since all proofs are constructions of witnesses, they are executable as programs. Program code may be extracted from the Coq tool automatically into the programming language OCAML. The language OCAML is an object oriented version of ML. It is at the same time the implementation language of the Coq system.

Another general advantage of Coq is that its type system is slightly more powerful than the simple type theory that is the basis for classical HOL. The type system of Coq enables dependent types. In fact, the type system has to be more powerful as Coq uses the Curry-Howard isomorphism: dependent types are used to represent universal and existential quantification. Yet the type system is sufficiently restricted such that it remains decidable.[2] However, there are some constructions that are included in Coq that violate the decidability: pattern matching constructions over dependent types can become undecidable.

Using the Curry-Howard isomorphism necessitates the use of a more powerful type system to represent quantification, as we have seen above. Besides complications with respect to type checking, the increased strength of the type system pays off with respect to expressivity. Dependent types are known to be the natural logical representation of modules, see for example [MQu86]. Although in principle it is very simplistic to assume that any module can be reflected into the logic as a dependent type, it is certainly true that for structures that are of interest in the domain of reasoning it is very desirable to be able to represent them as first-class citizens of the logic [Kam99a]. A very good example for such application domains of interactive theorem provers, where modular structures are reflected into the logic, is abstract algebra, e.g. [KP99]. Dependent record types are the natural representation of modular structures. Coq offers dependent records and, since recently, additionally an independent module system [Chr03].

Coq's original foundation, the Calculus of Constructions, has been specialized to a calculus of *inductive* constructions [CP90] that includes inductive definitions. Similar to a datatype definition as in a programming language like ML, an inductive definition in Coq is defined by a set of rules describing the signature of the

[1] With *tnd*, a contradiction to $A \wedge \neg B$ is equivalent to $\neg(A \wedge \neg B)$, which is equivalent to $\neg A \vee B$.

[2] In general, type checking for dependent types is undecidable because a type could, for example, represent the Halting problem. Coq's typed λ-calculus, the language for types and terms, is not Turing-complete: because of the typing constraints, no general recursion combinator can be expressed. There is a fixpoint-combinator in Coq, but it expects a termination proof for the recursive function given to it as an argument. Similar to classical HOL, all functions are total and hence do not cover the full class of μ-recursive functions.

constructors of the type. However, Coq's logic is, according to the Curry-Howard isomorphism, defined by its types. Therefore, an inductive definition may as well be used to define logical formulas. Examples for this additional feature are given in Section 3. Different to classical datatype definitions, an inductive definition does not even enforce a base case. Using inductive definitions for the formalization of computer science related subjects is very natural because types defined by an inductive definition automatically contain an induction principle and so-called exhaustion properties that enable to reason by *inversion*: if we need to show a property for all elements of a type, it suffices to make a case analysis over all different manifestations of elements of the type given by the constructors of the inductive definition.

In the early days, the usability of the Coq system was inferior to other HOL systems, but in the meantime techniques like simplification tactics that have proved successful in other tools have been integrated into Coq. The expressivity of Coq and its code extraction properties make it an interesting tool for the creation of generic frameworks, i.e. abstract formalizations of practical specification and verification problems that may be instantiated to various applications.

An excellent introduction to working with Coq is [BC04]. More concrete features of the Coq system will be explained when we use them in the following exposition.

2. Formalizing Java Security

The Java programming language has been analyzed with various formal methods. A comprehensive survey comparing different approaches has been published as early as 2001 [HM01]. In the meantime, these early efforts have been partly accomplished.

Type Safety Analysis

Quite a few of these formal approaches are based on interactive theorem proving analysis. Some of these projects have addressed type safety of the Java language, e.g. [ON99, vOh01]. Type safety is proved by an analysis of the static type system of Java. Basically, these approaches formalize the Java language with its type system and semantics and then show that well-typed programs do not go wrong. The notion of "do not go wrong" has to be further refined. It depends on the semantical model that is used. In a so-called *small-step* semantics [Mos99], we can explicitly reason about termination; there, "do not go wrong" corresponds to "a well-typed program does not get stuck". This informal notion again means that a well-formed program always reaches a value. Here, the notion of type safety is entailed in the properties *progress* and *preservation* as follows. A well-typed term can take a further step, according to the reduction rules of the semantics, or it is a *value* (progress), and if a well-typed term takes a step of evaluation, then the resulting term is also well-typed (preservation, sometimes called subject reduction) [Pie02].

By contrast, a *big-step* semantics (as, for example, used in [ON99, vOh01]) presupposes termination when defining the evaluation rules of the structural operational semantics. Preservation is the same, but progress differs as we already assume that a step can be taken.

The first step to prove the security of a programming language is to ensure that it is type safe. We consider the type safety of mobile code of Java, the so-called bytecode, which is an intermediate code run by the Java Virtual Machine.

Java Virtual Machine

Java's run time environment is called the Java Virtual Machine (JVM). Original source code of Java is compiled into bytecode, a format that can readily be interpreted by the virtual machine.

One of the main ideas of virtual machines is to support mobile code by providing an interpreter for any platform on which the language might be run. Thereby, platform-independence and portability of code is granted. The concept of a virtual machine dates back to the P-code of Pascal [Wir76, Nel79]. A virtual machine is not only found in Java and its dialect for Smart Cards called Java Card [Che00]; it has also been adopted by Microsoft's .NET framework in form of the Common Language Runtime System (CLR). The concept of a bytecode-based intermediate language has caught on – there are now even compilers to Java bytecode from other languages, for example C++.

Another function of a virtual machine is to control the interaction between mobile code, so-called applets, and the local run-time environment in order to prevent attacks from malicious applets.

Besides access control through stack inspection and sand-boxing of dynamically loaded classes, one security function of the virtual machine's architecture is a so-called *bytecode verifier*. It performs a static analysis, i.e. a type check, on a given bytecode program. The verification process entails the assurance that the applet's bytecode does not attempt to perform ill-typed operations at run-time that would undermine the access control of the JVM. Potentially insecure programs are detected in the verification process prior to execution and can be rejected. The main properties that are excluded in the verification are forging object references from integers, illegal casting of an object reference from one class to another, calling directly private methods of the application programmers interface (API), jumping in the middle of an API method, or jumping to data as if it were code [Ler03]. However, in order to assure that the security functions of the bytecode verifier do what they are supposed to do, a bytecode verifier must be verified itself. The verification of such a bytecode verifier can be based on type systems. This verification has to show that the implementation of the verifier implements the type check given by the type system. As the typing rules can be many and intricate, such a verification is error-prone. It is a very well-suited application for a machine-assisted verification, the more so as bytecode verifiers are quite likely candidates for high-level Common Criteria evaluations. The Common Criteria are a standard for security [CC05]. The Common Criteria requirements have complex dependencies, but in the higher evaluation assurance levels (EAL) of these criteria, formal methods are a sensible means to achieve a certificate. The strictest level is EAL7, where a formal representation of the high-level design and formal proofs of correspondence with the security requirements must be provided.

Klein and Nipkow [KN02] report on their verification of Java's bytecode verification using Isabelle/HOL. Barthe and Dufay provide a similar work in Coq but they use the following different approach of decomposing the JVM [BD04] similar to [SSB01].[3] As we have seen above, a virtual machine consists of the bytecode verifier and an interpreter. The two parts of the original JVM are named *type* (or abstract) machine for the verifier, and *offensive* machine for the interpreter. In order to simplify the intricate proof obligations, the task of verifying a virtual machine is reduced to a combined so-called *defensive machine*. The defensive machine is a theoretical combination of the offensive and the type machine, i.e. an interpreter that performs type checking. The type safety verification on a defensive machine is simpler to perform as the typing information is contained in the definition of the interpreter, i.e. a defensive machine corresponds to a typed operational semantics.

Bytecode Verification in Coq

The method implemented in the so-called Jakarta framework in Coq [BD04] supports the process of combining the two parts of the JVM to simplify the verification.

The applied method enables reasoning about the type safety of the defensive operational semantics. However, to validate that the results actually match with the actual JVM, one has to show that the defensive machine is implemented by the combination of offensive and type machines. This proof task is called cross-machine validation. More precisely, this task consists of proving the following property. The offensive and defensive machines must coincide on those programs that are accepted by the type virtual machine. This property can be shown in two parts:

- The offensive and defensive machine coincide on all programs that do not raise a type error in the defensive machine.
- Every program that raises a type error on the defensive machine also raises a type error on the type machine.

Practically, the two proofs above are equivalent to showing that the offensive and type machines are sound abstraction of the defensive machine. We concentrate in this paper on the second bullet point. The first part has been done as well (see [Kam06]).

If the cross-machine validation is performed successfully, a bytecode verifier can be extracted from the Coq formalization of the type machine. The advantage of Coq, its module concept and code extraction mechanism is that a bytecode verifier can be defined by the output signature of a functor that demands as

[3] Some names differ from [SSB01]: the offensive machine is called trustful, for example.

input all needed items and corresponding proofs as sketched above. Reuse of Coq formalizations and proofs is actually possible. As we will see in the following, we can reuse the module called BCV [Duf04] that enables a general transformation from an abstracted type system to a bytecode verifier.

The Jakarta framework provides some automated support of the sound abstractions needed as provisos for the generation of the bytecode verifier. Given the specification of the type and offensive machine, it tries to find the right abstraction functions and set up the soundness proofs.

The Bicolano language [Bic07] produced in the Mobius project [Mob07] is a more recent attempt to formalize the Java bytecode language and semantics in Coq. It can be seen as an extension of the Jakarta framework towards a standardized language and program semantics for Java bytecode. The proof results of Jakarta are reusable for Bicolano and Mobius. In the Mobius project the concept of *proof-carrying code* of Necula and Lee [NL96] is implemented for Java bytecode. Proof-carrying code is code that carries proofs of program properties that have been mechanically verified by a proof assistant. Thereby mobile code can be automatically verified on the end users' devices with respect to properties that have been interactively proved by the software provider.

The subject of the work presented in Section 3 is to complement the bytecode analysis to other security properties than just type safety. We consider the classical property of *non-interference*. For the reasons of brevity of the formalization, we restrict the attention merely to security types, leaving out classical type information. It is feasible to extend rather than complement a classical bytecode verifier by security types using a simple combination of classical types – like nat, bool, etc – with the security types and constructing corresponding combined defensive and type machines. This combination is deeply rooted in the core functions of the operational semantics and cannot be simply modularized.

Non-Interference

Non-interference is an abstract formal description of confidentiality. Confidentiality is besides integrity and availability one of the three properties that are generally used to define security. Non-interference has first been devised by Goguen and Meseguer in [GM82]. Based on abstract notions of a security policy and information flow, it defines that no confidential information may flow to public parts of a system.

We use a state-based version of non-interference. That is, we define a concise abstraction of all relevant run-time information of an execution state sufficient to express information flow, or rather the absence of it. Therefore, we assume partition of data in high (H) and low (L) partitions. This partition is the security policy that we consider as parameter to the system. Other work using state-based notions of non-interference exists, e.g. von Oheimb's Non-Leakage [vOh04]

Informally, non-interference now means that no information can flow from H to L — an attacker can infer no H-information from the output of L-variables.

More precisely, we define first an indistinguishability relation $\sim$ on states as follows. H-states are all indistinguishable, but L-states must be equal to be indistinguishable. Let further $s \Downarrow u$ be an evaluation relation on states that holds if s is an initial state, u is a final state and s evaluates to u. Now we can define non-interference formally.

Definition 2.1 (Non-Interference). For all states s, s', u, and u' such that $s \Downarrow u$ and $s' \Downarrow u'$, if $s \sim s'$ then $u \sim u'$.

The property of non-interference is the goal that we are going to prove for a given type system for Java bytecode in Section 3. This kind of proof is usually called correctness of the type system as it shows that a term that is well-typed does have the property encoded by the type system, here, for example, non-interference. This property is also the subject of type systems that describe information flow security and that have been the source for the type system we consider.

Information Flow Security

In early work on confidentiality of data in computer programs, Fenton [Fen73] and Bell and LaPadula [BLP73] developed *mandatory access control*: each data expression is labelled with a security label. Security control consists in a simultaneous computation of the dynamic flow of information with the running program. Mandatory access control has proved to be too restrictive because of a phenomenon called *label creep*. As

this approach considers execution paths, labels increase monotonically in sensitivity throughout execution. The results of the security computation tend to be labeled too sensitively for their intended use.

The classical idea to statically analyze the information flow in a program to ensure security is due to Denning and Denning [DD77]. To give an intuition, consider the simple if-statement in the following pseudo-bytecode snippet.

```
1: if x_H = 0 then 4
2: y_L = 1
3: goto 5
4: y_L = 0
5: end
```

Assume for simplicity that both variables range over $\{0, 1\}$. If the high (confidential) variable $\mathtt{x}_H$ contains value 1, then the low (public) variable $\mathtt{y}_L$ holds 0 at the end in line 5, and vice-versa: if $\mathtt{x}_H$ is 0, then $\mathtt{y}_L$ contains 1. Hence, there is an information flow from $\mathtt{x}_H$ to $\mathtt{y}_L$. This kind of information flow is called an *implicit flow* in contrast to an explicit flow, as, for example, given by a direct assignment $\mathtt{y}_L \mathtt{:=} \mathtt{x}_H$.

This is the basic idea of illegal information flows from high (H) to low (L), for which the Dennings defined general rules for analyzing program code. The analysis with such rules appears to be a tedious task for a human and therefore an ideal candidate for computer support. One approach to introduce machine assistance into the static analysis process is to provide type systems for information flow security. Actually, there is a whole bunch of different type systems for information flow analysis. A comprehensive overview is [SM03].

Basically, an information flow type system for security encodes non-interference in a specific set of types that are described by a set of typing rules. It is not surprising that it is possible to encode logical information in types because this is exactly what HOL is all about.[4] However, the difference in the intuition of type systems here is that they are intended to be decidable. A type system is supposed to be used for static analysis before compilation or interpretation of a program to automatically filter out incorrect code. This is the same idea that is usually connected to static analysis of programs and type safety, as sketched above.

An information flow type system encoding non-interference can be used to extend the Java bytecode verification by replacing the original type system with the information flow type system.[5] Basically, such a type system consists of security-types (simply $\{L, H\}$) that are assigned to all program variables and to all program points.

In the type system approach, every program expression has a security type that is completely static: unlike mandatory access control mechanism, these security labels are not computed at run time. Security is enforced by type checking; the compiler reads a program containing security types and by type-checking ensures that the program does not contain improper information flows. In addition to the types for program expression, type systems for information flow analysis use *type environments* that are assignments from program points to security types (usually H and L). These program-counter labels correspond to the dynamic process sensitivity labels used in mandatory access control. They enable to exclude implicit information flows as they can mark program *regions* as security sensitive (high), for example, the branches of an `if` that is controlled by a high program variable. The major advantage of type environments compared to dynamic analysis is that the label creep may be avoided. Dynamic enforcement uses only information about single execution paths. Therefore, once a security label has crept up, for example after an if-statement, the label has to stay up on the execution path. In a type environment, all possible execution path are reflected in the type assigned to a program point. Hence, it is possible, for example after an if-statement, to downgrade a program-counter's security label, for example if it has been low before the `if` and the compile-time type checking can prove that no possible execution path through the `if` contains an insecure assignment.

The meaning of an information flow type system is, similar to classical type systems, that if a program is well-typed in the non-interference type system, then it is secure. A program being secure means it is non-interfering: an attacker cannot extract high information from low outputs. Staying with the example above: a security policy that assigns the levels H to x and L to y must assign the level H to all program

[4] In fact, HOL, the simple theory of types, descends from Russell's Theory of Types. It was Russell's original aim to overcome the inconsistencies of naïve set theory by introducing the notion of types representing the domains of predicates in a strict hierarchy.

[5] Clearly, the information flow type system has to entail the former classical type system in order to preserve classical type safety.

points $\in \{1, \ldots, 4\}$ to guarantee non-interference for the code snippet. Otherwise an attacker could learn the value of the confidential variable x from the public variable y.

An example for a type system for non-interference for a subset of the Java language is [BBR04]. It is based on the general concepts of [BN03] using stack types to control information flow.

However, not all works concerning the verification of non-interference are based on type systems and syntactical notions of non-interference. Joshi and Leino [JL00] base their work on a semantic notion of security. Using a program HH (*havoc on h*) that sets all high values with arbitrary values, security of program S can be defined as $HH; S; HH \doteq S; HH$ where $\doteq$ denotes program equality based on total correctness and ; sequential composition in a relational model. This notion encodes that the results of S on low variables remain the same independent of the initial assignment to high variables. Intuitively, this is the same as the "low-equality" expressed by non-interference. Joshi and Leino argue that this semantic approach grants more flexibility than the type-based approaches. Most prominently, they show that their characterization is more precise (some programs that are secure and are rejected by type-based approaches are accepted here) and there is more flexibility as their notion of security applies to any program construct that has a semantics, unlike in type systems where the rules have to be extended for each new construct. They illustrate this flexibility quite impressively by applying their approach to nondeterminism and exceptions. However, as the authors also show, these advantages have their price, most importantly, various fixpoints characterizing recursion and loops have to be found. These are severe limitations for static analysis as they limit automation.

Non-Interference in Coq

The idea to use Coq for proving non-interference for the JVM is to simulate the type system [BBR04] by a *typed* operational semantics, i.e. a defensive machine. That is, the typing rules of the type system are integrated into an evaluation function. For this typed operational semantics, we define non-interference. Using Coq, we prove non-interference for the defensive machine. That is, we prove that for all executions an attacker cannot learn anything about high data.

From the defensive machine we can then abstract a type machine and an offensive machine in Coq. The type machine represents a security bytecode verifier for the JVM, i.e. a type checker that assures that a program that is accepted does not have illicit information flows. The input to this security type verification is a security policy consisting of an assignment of all program variables and program points to $\{H, L\}$, the security types.

Other work formalizing non-interference for Java in theorem provers includes the works by Naumann [Nau05] and Strecker [Str03]. In difference to the work described here, they address Java source code. Another work that formalizes also Java bytecode is [ACL03], which considers isolation properties in Java Card. Their work seems to address a larger fragment of the JVM than ours but they consider a very restrictive scenario where there is no information flow between contexts.

3. Proving Non-Interference of Bytecode in Coq

In this section, we report on the formalization that the author performed at INRIA Sophia-Antipolis in collaboration with Gilles Barthe in the Everest team [BK05]. The main ideas and some general concepts of Coq have already been introduced in Section 1. We introduce now step by step the concrete features of Coq as they arise when presenting the model. In this paper, we present a simple bytecode language that can be seen as a subset of Java bytecode – which is basically a simple imperative language. The goal of the work is to concentrate on the modularity of the specification and proof of non-interference. Extension of this approach to object-oriented features, i.e. objects and methods, works in principle but we encounter the usual difficulties with aliasing and nested calls. We summarize our experiences in the last paragraph of this section.

Axiomatic Framework

The formalization makes use of the Coq infrastructure of modules to provide a framework. This means that parts of specification and proof are performed in a generic manner.

More precisely, we specify on one side basic notions like program points, control dependence regions, and operand stacks in modules that can be reused. On the other side, we provide a functor that specifies in its input signature the constructors and their properties that are needed to derive non-interference.

The actual language that we analyze is a small subset of the Java language without objects and methods. It is the same subset that is also considered in [BBR04]. Although this might seem not representative, it is for the demonstration of a Coq application sufficient. Moreover, the genericity of the provided framework is such that it can be applied to larger portions of the language. This is the axiomatic framework idea that can be realized using modules: we provide a generic formalization of non-interference for bytecode languages. The actual language resides in a separate module and hence is exchangeable. The framework can be applied to different bytecode languages, and the axiomatic framework automates the non-interference proofs. The use of *axiomatic* here does not mean that we assume anything without proof. It just means that the interfaces to the actual language under consideration are given by axioms of a signature of Coq functors.

The global goal of non-interference is defined by a signature as follows.

```
Module Type NON_INTERFERENCE.
  Parameter 𝒮: Set.
  Parameter ∼ : 𝒮 → 𝒮 → Prop.
  Parameter =se: 𝒮 → 𝒮 → Prop.
  Parameter ⇓: 𝒮 → 𝒮 → Prop.
  Axiom NonInt: ∀ s s' r r': 𝒮,
                s ∼ s' ∧ s =se s' ∧ s ⇓ r ∧ s' ⇓ r' ⟹  r ∼ r'.
End NON_INTERFERENCE.
```

The header `Module Type` indicates that the following block of Coq code is a signature which can be thought of as the type of a structure (a structure would be introduced simply by `Module`). A signature may also be used to specify the target type of a functor. A functor transforms structures, i.e. modules, according to its source signature into ones according to its target signature. The constants declared as `Parameter` with their respective types represent the type of states $\mathcal{S}$ and predicates over states. The types `Set` and `Prop` are two distinct predefined Coq types that form the basis of the type hierarchies of specifications and propositions. The arrow $\rightarrow$ is the function type constructor. The predicates of the signature `NON_INTERFERENCE` are an indistinguishability relation $\sim$, an equality on the security environments representing the security policy $=_{se}$, and a big-step evaluation relation $\Downarrow$, i.e. `s` $\Downarrow$ `r` means that `s` is an initial state and evaluates to final state `r`. The equality `s` $=_{se}$`s'` is a predicate that defines that the security environments of the two states `s` and `s'` are identical. The definition of such a separate equality on states enables the abstraction of program points here. Concretely, a security environment is given by an assignment of program points to security levels.

The `Axiom` in this functor is the definition of non-interference as introduced in the previous section with the explicit constraint that the security policies of the states have to be identical.[6] Although the main theorem is an "axiom" this does not mean that it is blindly assumed. On the contrary, the functor concept necessitates that this proof obligation has to be solved by a module that has this result type. The arrow $\Longrightarrow$ is the Coq implication for the type `Prop`.[7]

Modular Structure

The overall modular structure of the Coq framework is depicted in Figure 1. Graphically, we have emphasized the genericity of the non-interference proof by separating the structure in the middle. The goal of the axiomatic framework is declared in the signature `NON_INTERFERENCE` introduced in the previous paragraph. We explain the structural dependencies now before presenting the actual contents of the formalization.

The functor `NonInterference` provides a proof for the `Axiom NonInt` based on its input signature `REGION_PROG`. The latter signature is the interface declaring the goals for the functor `JVM_OpSem` which contains the definition of the defensive operational semantics. It is a small-step semantics defined by an evaluation function over a more concrete notion of state processing single instructions. This single-step defensive operational semantics is based on several inputs: the functor `Region` provides the necessary properties for

[6] This additional assumption is usually implicit in the definition of non-interference. The other part of the security policy, i.e. the security level assignment of variables, is left here as well implicit in the definition of indistinguishability.

[7] Actually, this arrow is identical to the function type constructor $\rightarrow$ because types are propositions in Coq. We use this syntactic sugar when we want to emphasize the logical content.

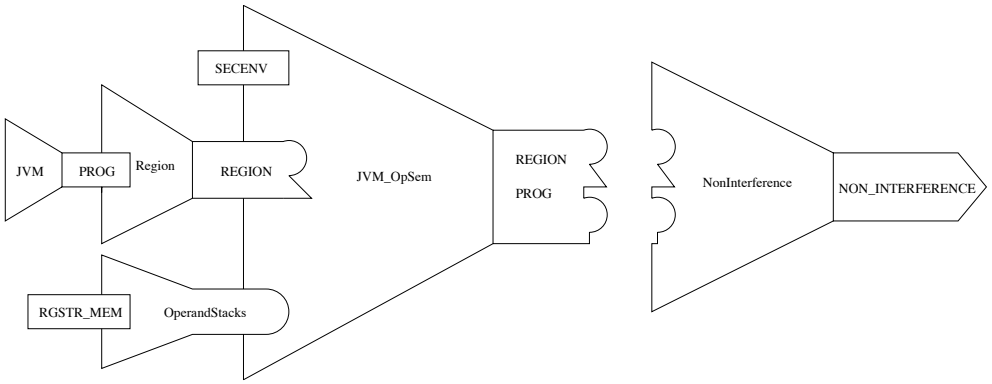

Fig. 1. Signatures (boxes), modules and functors (triangles) of axiomatic framework

control dependence regions according to the input signature `REGION`, the functor `OperandStacks` provides a notion and basic lemmata for operand stacks, and `RGSTR_MEM` specifies values, registers and an abstract security policy `env` for registers. The input signature `SECENV` specifies an abstract security policy `secenv` assigning H and L to program points. The security policies are left abstract because they are thought of as parameters to the entire specification. The entire process of security bytecode verification is relative to the security policies `secenv` and `env` representing the security type of a program. The syntactic definition of the programming language is contained in the module `JVM` that is inserted into the region functor by its input signature `PROG`.

In the following, we introduce in detail the major parts of this axiomatic framework.

Defensive Operational Semantics

Besides the notions of regions, the functor `JVM_OpSem` defining the defensive machine imports, via the functor `Region`, the syntax of the bytecode language we want to investigate. It also loads a theory of operand stacks including basic lemmata.

The syntax of the bytecode is defined as instructions in a simple inductive definition (see below). For all possible instructions, we specify their parameter types. A `push` takes a natural number of type `nat` that is to be pushed on the stack, `load` and `store` are parameterized by a register location, given by the type `locs`, and `goto` and `ifthenelse` take program points representing the jump point. The latter `ifthenelse` is the simple assembler instruction: if the top of the stack is 0, then goto the next line else go to the jump point. The remaining instructions `nop`, `iadd`, and `halt` take no parameters.

```
Inductive instr : Set :=
    nop  : instr
  | push : nat → instr
  | iadd : instr
  | load : locs → instr
  | store: locs → instr
  | goto : 𝒫 → instr
  | ifthenelse   : 𝒫 → instr
  | halt : instr.
```

This non-dependent inductive definition strongly resembles an ML datatype. Coq also provides automatically induction and inversion rules for any inductive definition facilitating the use of such definitions.

A program is a function binding instructions to program points.

```
Definition program := 𝒫 → instr.
```

The one step control flow relation $\mapsto$ is also defined as a simple inductive definition for a fixed but arbitrary

program constant p in the following expression. We just show the first clause; there are similar ones for the other instructions.[8]

```
Inductive ↦: 𝒫 → 𝒫 → Prop :=
    step_goto: ∀ pc l: 𝒫, p pc = goto l ⟹  pc ↦ l
  | ...
```

Given the syntax of a programming language as specified in signature PROG, the functor REGION passes this part of the specification on to the functor JVM_OpSem that defines the operational semantics.

In JVM_OpSem, we define first the notion of states as a record with four fields: a register binding rm, an operand stack os, a program point pc , and a security environment se.

```
Record 𝒮 : Set :=  { rm: env; os: stack; pc: 𝒫; se: secenv }.
```

A record type in Coq has projection functions automatically associated to it. For example, with se s we can annotate the security environment component of a state s. Coq generates for every record a constructor with the name convention Build_*record-name*, e.g. the constructor for states is Build_$\mathcal{S}$. Taking into account that records are a central feature in Coq's rich type theory, this is comparatively weak, considering that other HOL tools, e.g. Isabelle, have an elegant and intuitive infix constructor for records. Dependent records are possible in Coq. They are useful for defining structures that entail proofs as components.

Next, we define the small-step defensive semantics. We are going to define a typed evaluation function over states. As we want to accommodate type errors in this execution function, we first define a second type of return states $\mathcal{S}_\mathcal{R}$ as target type for the evaluation function including a constructor that passes on the previous state for sound executions, constructors for the errors, and a constructor to finalize an execution.

```
Inductive 𝒮ℛ: Set :=
    Normal : 𝒮 → 𝒮ℛ
  | Abnormal: 𝒮ℛ
  | SecError: 𝒮ℛ
  | Result: 𝒮 → 𝒮ℛ.
```

There are two error cases: Abnormal is used for conventional type errors and SecError for errors concerning security.

The defensive single-step execution function can now be defined as a function from $\mathcal{S}$ to $\mathcal{S}_\mathcal{R}$ in a case analysis using Coq's pattern matching device match over the structure of the current states instruction. The variable p is a parameter of the current module and denotes a program. Hence, the current instruction is p cpc, where cpc is the current state's program point pc s.

```
Definition dexec (s:𝒮) : 𝒮ℛ:=
   let crm := rm s in
   let cos := os s in
   let cpc := pc s in
   let cse := se s in
   let cin := p cpc in
   match cin with
     nop     ⇒ Normal (Build_𝒮 crm cos (succs pc) cse)
   | goto j  ⇒ Normal (Build_𝒮  crm cos j cse)
   | push z  ⇒ PUSH z crm cos cpc cse
   | load l  ⇒ LOAD l crm cos cpc cse
   | store l ⇒ STORE l crm cos cpc cse
   | ifthenelse l ⇒ IFTE l crm cos cpc cse
   | iadd    ⇒ IADD crm cos cpc cse
   | halt    ⇒ HALT crm cos cpc cse
   end.
```

The cases where the current instruction is nop and goto j for some program point j are simple. For nop, nothing happens to the state other than increasing the current program point to the next one using the successor function succs on $\mathcal{P}$. The goto instruction has a similar effect just setting the program counter to j.

A more difficult example is store l, the instruction that stores the top of the stack in register l (see below). If the current operand stack cos is empty, identified by the constructor e_st of stacks, we have an

[8] We use here infix syntax as it is possible in most HOL tools for clarity of exposition different to our original Coq sources. In principle, mixfix syntax is possible in Coq but it is difficult to use.

execution error. If the stack consists of the stack `ros` with top element `v`, i.e. `p_st v ros`, we compare the security level of the current program point with the security level of the register `l` in which the top of the stack `v` shall be stored. The level of the former has to be less than or equal to the latter, which is checked by the predicate `lelev` defined on the security levels $\{H, L\}$. In the positive case, we create a normal return state by updating the register binding `crm` and increasing the program point. If the level of the current program point is greater than the level of the register, i.e. `lookup_se cse cpc` is H and `lv (lookup crm l)` is L, we would have an illicit information flow. Hence, we return the error `SecError`.

```
Definition STORE (l: locs)(crm: env)
                 (cos: stack)(cpc: 𝒫)(cse: secenv) :=
    match cos with
      e_st ⇒ Abnormal
    | p_st v ros ⇒
        match (lelev (lookup_se cse cpc) (lv (lookup crm l))) with
          true ⇒ Normal (Build_𝒮 (update crm l v) ros (succs cpc) cse)
        | false ⇒ SecError
        end
    end.
```

The other cases of the defensive execution function assert in a similar way type correctness and absence of information flow. For the full definition of the function, we refer to the author's web page [Kam06].

Proving Single-Step Non-Interference

The functor `JVM_OpSem` containing the definition of the defensive machine provides now the lemmata for the small-step semantics describing the preservation of non-interference for single-step executions needed as provisos for the global inference of non-interference in `NonInterference`.

Non-interference is about indistinguishability. The indistinguishability relation has to be defined for states. As states are made up of various parts, we define indistinguishability in several steps. Values are defined in the signature `RGSTR_MEM` as a record containing an abstract value and a level ∈ { `high, low` }.

```
Record val: Set := { iv: value; lv: level }.
```

The corresponding notion of value indistinguishability is given by the following inductive definition defining high values as generally indistinguishable and low ones only if they are equal.

```
Inductive val_in: val → val → Prop :=
    val_in_high: ∀ v v': value,  val_in (Build_val v high) (Build_val v' high)
  | val_in_low: ∀ v v':value, v = v' ⟹  val_in (Build_val v low) (Build_val v' low).
```

Similarly, operand stacks need an indistinguishability relation. Theirs comes in two parts. High stacks are stacks whose members have all level `high`.

```
Inductive high_st: stack → Prop :=
    high_e_st: high_st e_st
  | high_p_st: ∀ v:value, ∀ s:stack, high_st s ⟹  high_st (p_st (Build_val v high) s).
```

If two stacks are high, they are indistinguishable as specified in the following predicate `os_in0`.

```
Definition os_in0 (s s':stack) := high_st s ∧ high_st s'.
```

In general, stacks are only indistinguishable if their top elements and the remainders of the stacks are indistinguishable.

```
Inductive os_in: stack → stack → Prop :=
    os_in0_os_in: ∀ s s': stack, os_in0 s s' ⟹  os_in s s'
  | os_in_cons: ∀ s s':stack, ∀ v v':val,
                os_in s s' ⟹  val_in v v' ⟹  os_in (p_st v s) (p_st v' s').
```

For register bindings, the indistinguishability is extensionally defined over all contained values.

```
Definition rm_in (rho rho':env): Prop :=  ∀ l: locs, val_in (lookup rho l) (lookup rho' l).
```

With these preparations, we can define the indistinguishability relation $\sim$ on states.

```
Definition ∼ (s s': 𝒮):Prop :=  rm_in (rm s)(rm s') ∧ os_in (os s) (os s').
```

To show non-interference for arbitrary executions, we prove two lemmata that consider the preservation of indistinguishability during single-step execution. This way of reducing non-interference to those two properties is a classical one described in [Rus90] and also already in [GM82]. There, they are called unwinding rules.

The first one addresses two parallel single-step executions of `dexec`. It assumes two indistinguishable states `s` and `s'` that have the same security environment and are at the same program point. The relation $=_{pc}$ is equality on states with respect to program points $\mathcal{P}$. If those two states `s` and `s'` are executed by the defensive execution into normal return states, then the resulting states are also indistinguishable.[9]

```
Lemma llni: ∀ s s' u u': 𝒮,
  s ∼ s' ∧ s =pc s' ∧ s =se s' ∧ dexec s = Normal u ∧ dexec s' = Normal u' ⟹ u ∼ u'.
```

The other case that is of interest is that we have a current state `s` whose security environment `se s` assigns a `high` to the current program point `pc s`. If the entire operand stack of the current state is high and the current state `s` evaluates without error to the state `u` in one step of the defensive machine, then the previous and the next state are indistinguishable.

```
Lemma hlni: ∀ s u: 𝒮, se s (pc s) = H ∧ high_st (os s) ∧ dexec s = Normal u ⟹ s ∼ u.
```

In other words, `llni` expresses that *two parallel* single-step executions from the same program point preserve indistinguishability for the post-states. The second lemma `hlni` expresses that in *one execution thread* in a high state indistinguishability is preserved between pre-state and post-state. A high state is one whose security environment renders high on the current program point and the entire stack is filled with high values. Intuitively, two threads in low states stay indistinguishable only when they move synchronously to next states (`llni`), whereas once they creep up to high they may move independently (`hlni`). In this manner, the two main lemmata are used when proving non-interference by lifting indistinguishablity from the single-step `dexec` to the many-step evaluation ⇓ in the following section.

There are a few other lemmata needed. The whole set of necessary assumptions for the global proof of non-interference is contained in the input signature `REGION_PROG` of the functor `NonInterference`. This signature is in full contained on the author's web page [Kam06].

Proof of Non-Interference

Based on the properties of signature `REGION_PROG`, the proof of non-interference is constructed in the functor `NonInterference`. This manifests itself in that the target of the functor is declared by the signature `NON_INTERFERENCE` (see above). Technically, in Coq a functor is realized by parameterizing a module with a parameter list and a return signature. For our example, the first line of the module containing the global non-interference proof is as follows.

```
Module NonInterference (P: REGION_PROG) <:  NON_INTERFERENCE.
```

Practically, when developing a specification and a proof interactively such a header opens up a local proof context. In the following, we can assume the contents of the parameter structure `P` as specified in the signature `REGION_PROG` and base our current specification and proof on it. For example, we can now use our lemmata `llni` and `hlni`, seen above, by referencing them with `P.llni` and `P.hlni`. Once we end our session by typing `End NonInterference`, Coq will check whether the output signature `NON_INTERFERENCE` is matched. If all proofs in the local context are finished and no assumptions remain open, the prover accepts the functor and produces the corresponding abstract proof objects. Any time afterwards, we can reuse this functor as a generic proof: by supplying a concrete structure, e.g. `prog_lang`, of signature `REGION_PROG`, we can literally apply the functor as `NonInterference(prog_lang)` to generate the global non-interference proof for an arbitrary bytecode language `prog_lang`.

Technically, the big-step non-interference proof is performed in two steps. First, we introduce an n-step defensive semantics inductively.

```
Inductive evalsto : 𝒮 → 𝒮 → nat → Prop :=
    evalsto_res: ∀ s: 𝒮, result s ⟹ evalsto s s 0
  | evalsto_step: ∀ s s' s'': 𝒮, ∀ n: nat, exec s s' ∧ evalsto s' s'' n ⟹ evalsto' s s'' (S n).
```

[9] We use here ∧ to accumulate assumptions. In the original file, the ∧ and the ⟹ are all represented as function arrows →, which is logically equivalent (see Footnote 7).

The relational expression `exec s s'` has to be thought of as `dexec s = Normal s'`. The abstraction provided by the module concept is such that we can forget unnecessary detail here at the abstract level. Technically, `exec` is defined in the interface signature `REGION_PROG` as a predicate over states. The instantiation with `dexec` is realized at the end of the functor `JVM_OpSem` by the following function definition.

```
Definition exec := fun s rs: S ⇒ dexec s = Normal rs.
```

Using case analysis in a simultaneous induction over the two natural numbers `n` and `n'`, we can prove a main lemma for n-step execution.

```
∀ n n': nat, ∀: s s' r r': S,
          s ∼ s' ∧ s =se s' ∧ s =pc s' ∧ evalsto s r n ∧ evalsto s' r' n' ⟹  r ∼ r'.
```

Now, we simply define our notion of execution ⇓ from initial to final state as

```
Definition ⇓ (s r: S) :=  pc s = start ∧ ∃ n: nat,  evalsto s r n.
```

and can then prove the global theorem.

Theorem 3.1 (NonInt). `∀ s s' r r': S,  s ∼ s' ∧ s =se s' ∧ s ⇓ r ∧ s' ⇓ r' ⟹  r ∼ r'.`

Extracting a Bytecode Verifier

Finally, as a preparation to extracting a bytecode verifier for non-interference, we need to construct a type machine from the defensive machine by abstracting away the computational content. First, we define the components of the abstract state. An abstract value consists only in the level $\in \{H, L\}$ omitting the value.

```
Definition tval := level.
```

Type stacks `tstack` and type register bindings `tenv` are the same as defensive stacks and register bindings but over `tval` instead of `val`. Now we construct a second notion of type state $\mathcal{S}_T$ and type return state $\mathcal{S}_{RT}$ similar to the defensive state but leaving out the computation information.

```
Record S_T: Set :=  { trm: tenv; tos: tstack; tse: secenv }.

Inductive S_RT: Set :=
    tNormal : S_T → S_RT  | tAbnormal: S_RT | tSecError: S_RT | tResult  : S_T → S_RT.
```

Then, we define a typed execution function `texec` in analogy to `dexec`, which actually performs execution just on the type states.

```
Definition texec (s: S_T)(cpc:pcs): S_RT :=
   let crm:=(trm s) in
   let cos:=(tos s) in
   let cse:=(tse s) in
   let cin:=(p cpc) in
   match cin with
     nop     ⇒ tNormal s
   | goto j  ⇒ tNormal s
   | push z  ⇒ tPUSH z cpc crm cos cse
   | load l  ⇒ tLOAD l cpc crm cos cse
   | store l ⇒ tSTORE l cpc crm cos cse
   | ifthenelse l ⇒ tIFTE l cpc crm cos cse
   | iadd    ⇒ tIADD cpc crm cos cse
   | halt    ⇒ tHALT cpc crm cos cse
   end.
```

The definition is very similar to that of `dexec`. The full definition is contained on the author's web page [Kam06].[10] We illustrate again just the case for store below.

In comparison to `STORE`, the case `tSTORE` is almost identical: just the lookup term differs in that the level has not to be revealed, because values are just levels and in the construction of the next state no new program point is added.

[10] For the exposition, we have factored out the cases in `texec` to show the similarity to `dexec` – the original sources differ.

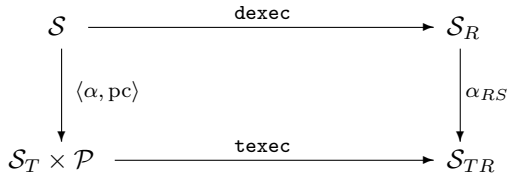

Fig. 2. abstracting the type machine

```
Definition tSTORE (l: locs)(cpc: P)(crm: tenv)(cos : tstack)(cse: secenv)
   match cos with
     te_st ⇒ tAbnormal
   | tp_st v ros ⇒
       match (lelev (lookup_se cse cpc) (tlookup crm l)) with
         true ⇒ tNormal (Build_S_T (tupdate crm l v)  ros cse)
       | false ⇒ tSecError
       end
   end
```

In the definition of the type states, the program point is left out, but in the type valuation function the current program point `cpc` is introduced again as a parameter to enable the reference to the program point in commands that need to respect the security environment.

The simple derivation of the type machine from the defensive machine is one of the advantages of the concept of defensive, offensive and type machine. It has the additional consequence that the cross-machine validation is rather trivial as well, as we will see next.

In order to prepare the proof of sound abstraction, we need abstraction functions to map a defensive state to a type state.

```
Definition alpha_S (s:S): S_T :=
  match s with (Build_S crm ros cpc cse) ⇒  Build_S_T (alpha_env crm) (alpha_stack ros) cse end.
```

The functions `alpha_env` and `alpha_stack` just lift the value abstraction to register bindings and stacks pointwisely.

Soundness of abstraction corresponds to showing that the diagram in Figure 2 commutes. This reduces to showing that every program that produces a type error on the defensive machine also produces a type error on the type machine. For the security type error, we prove the following lemma in Coq.

```
Lemma dt_commutes: ∀ s: S, dexec s = SecError ⟹  texec (alpha_S s) (pc s) = tSecError.
```

For the type error `Abnormal`, a similar lemma is proved. Finally, we can reuse the functor `BCV` from [Duf04] that given a type machine and the above commutation lemma generates a bytecode verifier by simple instantiation.

To finalize the cross-machine validation, we need to show the corresponding property for the offensive machine, i.e. that offensive and defensive machine coincide on all programs not raising type errors in the defensive machine. To this end, we can construct the offensive machine in a similarly simple fashion by deriving it from the defensive machine given offensive values that abstract from the security level. Again as a consequence, the proof of the needed commutation property is rather trivial (see [Kam06]).

Statistics and Lessons Learned

The entire formalization is about 2300 lines of Coq-Code (specification and proof): 1200 lines for the defensive operational semantics, 900 lines for the axiomatic framework (regions, operand stacks, and non-interference), and 200 for the abstraction of the type machine.

The type system we derive in Coq as a type machine corresponds to the information flow type system in [BBR04]. In principle, it is also possible to show the equivalence with this original type system formally in Coq. Therefore, one needs to define a second notion of non-interference and show that the offensive machine derived from the defensive machine is non-interfering with respect to this second notion of non-interference. However, as our notion of non-interference is similar to standard ones, this is of interest only if we want to mechanically verify the exposition of [BBR04].

The abstraction of the defensive machine to an offensive one can be performed in much the same way as the abstraction of the type machine. We just need to delete the type information and keep the computational information. An offensive machine corresponds to an interpreter for Java bytecode.

From a software engineering point of view, it is an interesting question how a non-interference framework would work in practice. This is not the subject of the work presented here. We believe that an iterative process of constructing the security type information is feasible. The idea is that a programmer would provide a list of program points and variables for which he demands H. The type checker can then assume L for everything else and can reject or accept. In the next iteration step, the type violations could be used to arrive at a better approximation until a good security policy is found that contains the programmer's initial requirements. Since an "all-H" assignment is always non-interfering, this simple procedure does terminate. It is an open question whether using type inference rather than just type checking together with possible monotonicity of typing rules may be employed to arrive at solutions that are minimal in sensitivity.

The use of modules in Coq, finally, is rather crucial to the entire framework development. It is the use of modules that enables reuse of entire specifications and proofs in an abstract manner. We can reuse the module `BCV` of [Duf04] and we can devise the structure of our axiomatic framework such that it may be instantiated to provide proofs and bytecode verifiers for other bytecode languages. However, Coq modules have the same deficiencies as usual: sharing is not possible and the name spacing facilities are not very sophisticated. For example, the specification of `regions` is given in the functor `REGION`. But as the security environment `secenv` also uses regions (because the lifting of a security environment lifts entire region's security levels), we need to share this part of a signature when both signatures `REGION` and `SECENV` are imported into `JVM_OpSem`. That is, although both of the imported signatures define their own abstract parameter `region`, this has to be identified now. It appears that in the current version of Coq's modules, it is not yet possible to explicitly set such sharing parameters. Coq produces two copies `S.region` and `R.region` for the two structures `S: SECENV` and `R: REGION`. As a work-around, we used the possibility to assume axioms to identify the two. It is quite disappointing to rely on such means. However, the question of sharing module parameters in general is more sophisticated as it addresses the question whether such shared parameters can or should be found automatically. There are cases, as above, where we want to unify similarly structured parameters further down in the import chain, but we surely do not want this to be done generally – otherwise one could not import different sets of parameters that coincidentally have the same type without them being identified. How to define a good measure for this and how to integrate it into a module system technically are interesting open questions for module development.

Extension to Object Orientation

The main question that remains to be considered is whether the approach we have presented scales up to object orientation. We have attempted to apply our modular specification and proof to the extensions with objects and methods (see the files `JVMo.v` and `JVMm.v` at [Kam06]). Corresponding extensions of the simple type system [BBR04] served as a role model. We exchanged the corresponding modules `JVM` and `JVM_OpSem` and adapted the proofs in the single-step operational semantics. Unfortunately, we could not scale up the approach. It is no problem to extend the formalization and proof of non-interference using a bijection between object heaps (already used in [BN03] at the source code level) as an additional part of the security policy. However, it is not any more possible to derive a simple type system in the process of cross-machine validation because the used bijection uses references to objects. Hence, no abstraction from the dynamic evolution of the object heap is possible any more when extracting the type system. The result would not any more be a simple type system suitable for static analysis. We currently investigate an additional abstraction from the concrete implementation of the object model based on pointers and heaps assigning security levels to classes. It enables a unified security policy for objects. The abstraction is inspired by current advances in the application of separation logic [TKN07] to similar pointer problems in type safety proofs.

4. Conclusions

In this paper, we have seen how interactive theorem proving with Coq can be used to prove information flow security of Java bytecode. We have introduced the basic ideas and concepts of Coq before we introduced the various building blocks of the formalization of Java security analysis. Based on the concepts of type systems

for information flow security, we have presented a modular formalization for analyzing non-interference for bytecode languages. We have applied this framework to a small subset of the Java bytecode language. The formalization is modular and can hence be applied to other bytecode languages by simple instantiation of our constructed functor. An abstract machine is derived from the defensive operational semantics. The formalization enables the automatic extraction of an executable bytecode verifier by applying an existing module to the abstract machine.

We have worked in Coq. Coq is not necessarily better suited than other HOL tools for the analysis of programming languages (see the work by Nipkow et al. in Isabelle, for example [ON99, KN02]) but we did profit from its module system. Modularity has proved useful in the example of the axiomatic framework for bytecode, where the entire formalization could be organized in a reusable way. If the functor for the defensive operational semantics is exchanged by a functor formalizing an operational semantics of an extended or different programming language, large parts of the framework can be reused. Reuse here should be read in the most general sense: functors merely have to be applied to provide specifications and proofs automatically. Also, the module `BCV` from an earlier framework could be reused to finally transform the derived type machine into a bytecode verifier. There are some minor technical flaws we encountered when using the module concept, as pointed out at the end of the precious section. The most part might be fixed in future releases of Coq but a remaining conceptual problem of modularity in general is how to find a sensible way of solving sharing of parameter structures or signatures. An interesting next experiment might be the transformation of the presented modular proof to other HOL tools with modules, e.g. Isabelle, and its concept of locales [Kam99a], in order to compare the module systems.

Acknowledgments

The author wishes to thank Gilles Barthe, Yves Bertot, Lucca Martini, David Naumann and Tamara Rezk for many helpful discussions while working on this formalization at INRIA Sophia-Antipolis or at conferences. I would also like to thank the anonymous referees that helped to greatly improve the paper.

References

[ACL03] J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. *Theorem Proving in Higher Order Logics, TPHOLs'03*. LNCS **2758**. Springer, 2003.

[Bic07] Bicolano and MOBIUS base logic. `http://mobius.inria.fr/twiki/bin/view/Bicolano`, 2007.

[BN03] A. Banerjee and D. A. Naumann. Stack-based Access Control for Secure Information Flow. *Journal of Functional Programming*, **15**(2):131–177, Cambridge University Press, 2003.

[BBR04] G. Barthe, A. Basu, T. Rezk. Security Types Preserving Compilation. *Verification, Model Checking, and Abstract Interpretation, VMCAI'04*. LNCS **2934**, Springer, 2004.

[BD04] G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. *Fundamental Approaches to Software Engineering, FASE 2004*. LNCS **2984**, Springer, 2004.

[BK05] G. Barthe and F. Kammüller. Certified Bytecode Verifier for Non-Interference. Technical Report, *INRIA Sophia-Antipolis*, 2005.

[BLP73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: A Mathematical Model. Technical Report MTR-2547(2), *MITRE Corp.* Bedford MA, 1973. Reprinted in *Journal of Computer Security*, **4**(2–3): 239–263, IOS Press, 1996.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'art: the Calculus of Inductive Constructions*. Springer, 2004.

[Che00] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, Reading, Massachusetts, 2000.

[CC05] National Institute of Standards and Technology. *Common Criteria for Information Technology Security Evaluation*. U.S. Dept. of Commerce, National Bureau of Standards and Technology. `http://csrc.nist.gov/cc`, 2005.

[Chr03] J. Chrzaszcz. Implementing Modules in the Coq System. In *Theorem Proving in Higher Order Logics, TPHOLs 2003*. LNCS **2758**: 270–286. Springer, 2003.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**(2): 56–68, Association for Symbolic Logic, 1940.

[CP90] T. Coquand and C. Paulin-Mohring. Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *International conference in computer logic, Colog'88*. LNCS **417**, Springer, 1990.

[Coq04] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, **20**(7): 504–513, Association for Computing Machinery, 1977.

[Duf04] G. Dufay. *Vérification Formelle de la Plate-Forme Java Card*. Thèse de Doctorat. Université de Nice Sophia-Antipolis, 2003.

[Fen73] J. S. Fenton. *Information Protection Systems.* PhD thesis, University of Cambridge, 1973.
[GM82] J. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of Symposium on Operating System Principles, SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.
[HM01] P. H. Härtel and L. Moreau. Formalising the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys (CSUR)*, **33**(4): 517–558, Association for Computing Machinery, 2001.
[How80] W. A. Howard, The Formulae-as-Types Notion of Construction. In Seldin and Hindley [SH80], pages 479–490.
[JL00] R. Joshi and K. R. M. Leino. A Semantic Approach to Secure Information Flow. *Science of Computer Programming.* **37**: 113–138, Elsevier, 2000.
[Kam99a] F. Kammüller. *Modular Reasoning in Isabelle.* PhD thesis, Computer Laboratory, University of Cambridge, Technical Report 470. August, 1999.
[Kam06] F. Kammüller. http://www.swt.cs.tu-berlin.de/∼flokam/coq/index.html.
[KP99] F. Kammüller and L. C. Paulson. A Formal Proof of Sylow's First Theorem – An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, **23**(3): 235–264, Kluwer Academic Publishers, 1999.
[KN02] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, **298**(3): 583–626, Elsevier, 2002.
[Ler03] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning, Special Issue on bytecode verification*, **30**(3-4): 235–269, Kluwer Academic Publishers, 2003.
[MQu86] D. B. MacQueen. Using Dependent Types to Express Modular Structures. In *Proc. of 13th ACM Symposion on Principles of Programming Languages, POPL'96.* Association for Computing Machinery, 1986.
[Mob07] Mobius: Mobility, Ubiquity and Security. `http://mobius.inria.fr/twiki/bin/view/Mobius`, 2007.
[Mos99] P. D. Mosses. Foundations of Modular SOS. In *Mathematical Foundations of Computer Science, MFCS'99.* LNCS **1672**, Springer, 1999.
[Nau05] D. A. Naumann. Verifying a Secure Information Flow Analyzer. *Theorem Proving in Higher Order Logics, TPHOLs'05*, Oxford 2005. LNCS **3603**, Springer, 2005.
[NL96] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI).* October 1996, pages 229–243. Operating Systems Review, Special Issue, ACM, 1996 and USENIX Association, 1996.
[Nel79] P. A. Nelson. A Comparison of Pascal Intermediate Languages. *ACM SIGPLAN Notices*, **14**(8): 208–213, Association for Computing Machinery, 1979.
[vOh01] D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.
[vOh04] D. v. Oheimb. Information Flow Control Revisited: Noninfluence = Noninterference + Nonleakage *9th European Symposium On Research in Computer Security, ESORICS'04.* LNCS **3193**, Springer, 2004.
[ON99] D. v. Oheimb and T. Nipkow. Machine-Checking the Java Language Specification: Proving Type-Safety. In Jim Alves-Foss (Ed.): *Formal Syntax and Semantics of Java.* LNCS **1523**: 119–156, Springer, 1999.
[Pie02] B. Pierce. *Types and Programming Languages.* Wiley, 2002.
[Rus90] J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report csl-92-2, *SRI*, Palo Alto, 1992.
[SM03] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, **21**: 5–19. IEEE Computer Society Press, 2003.
[SH80] J. P. Seldin and J. R. Hindley (eds). *To H. B. Curry: Essays on Combinatory Logic.* Academic Press Ltd, 1980.
[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.
[Str03] M. Strecker. Formal Analysis of an Information Flow Type System for MicroJava (extended version). Technical Report, Technische Universität München, July 2003.
[Tho90] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.
[TKN07] H. Tuch, G. Klein, and M. Norrish. Types, Bytes, and Separation Logic. In *Principles of Programming Languages, POPL'07.* ACM SIGPLAN **42**(1), Association for Computing Machinery, 2007.
[Wir76] N. Wirth. *Algorithms + Datastructures = Programs.* Prentice Hall, 1976.