Formal Methods for Smartcard Security

Gilles Barthe[1] and Guillaume Dufay[2]

[1] INRIA Sophia-Antipolis, France
Gilles.Barthe@inria.fr
[2] SITE, University of Ottawa, Canada
gdufay@site.uottawa.ca

Abstract. Smartcards are trusted personal devices designed to store and process confidential data, and to act as secure tokens for providing access to applications and services. Smartcards are widely deployed and their usage spans over several application domains including banking, telecommunications, and identity.

Open platform smartcards are new generation trusted personal devices with increased flexibility. Such devices, which benefit of increased connectivity and increased interoperability, can host several applets and allow new applets to be loaded post-issuance. Such an increased flexibility raises concerns about the possibility of logical attacks that could affect a very large number of devices, and requires the development of techniques and tools that can be used to increase the reliability of platforms and applications for trusted personal devices. The objective of this chapter is to describe some applications of formal methods to increase the reliability of smartcards and trusted personal devices.

1 Introduction

Smart cards are trusted personal devices whose characteristics are regulated by the ISO 7816 standard. As other trusted personal devices, smartcards are designed to store and process confidential data, and can act as tokens to provide users with a secure electronic representation in a large network. They are widely deployed and used in application areas such as mobile telecommunications, banking, transportation, electronic identity, and digital rights management (DRM). Further, they hold the promise to play a key role in the e-society, especially as a means to guarantee users a personalized, global, and secure access to applications and services.

The prominent role played by trusted personal devices in security sensitive applications make them an ideal target for attacks. Traditionally, the main concern with smartcards has been with hardware attacks in which the attacker gains access to confidential information or disturbs the functioning of the card through observation (e.g. of power or electro-magnetic radiations) or invasion (e.g. overriding sensors or attaching probes).

With new generation smartcards and trusted personal devices increasingly connected to networks and providing execution support for complex programs, the prospect of logical attacks has urged the trusted personal devices industry

A. Aldini, R. Gorrieri, F. Martinelli (Eds.): FOSAD 2004/2005, LNCS 3655, pp. 133–177, 2005.

to improve the quality of their software, as logical attacks are potentially easier to launch than physical attacks (for example they do not require physical access to the device, and are easier to replicate from one device to the other), and may have a much wider impact. In particular, a malicious attacker spreading over the network and disconnecting or disrupting devices massively could have devastating economic and social consequences and would deeply affect end users confidence in e-society. The Cabir virus which spread through Symbian cell phones during summer 2004, although it did not actually do any damage, sounded a strong warning that cell phone viruses may soon cause havoc if no appropriate security technology is developed. With the increasing use of voice over IP, the nightmare could also extend to all phone infrastructures.

The risk of devastating attacks on trusted personal devices justifies the development of methodologies and tools that increase confidence in the execution platforms that they support and in applications that are executed on-board such devices. The need for methodologies and tools is implicitly recognized by existing standards for evaluating security-sensitive IT products, such as the Common Criteria [31] which require the use of formal methods at its highest and most demanding levels EAL5-EAL7, and has triggered some substantial activity in the community of formal methods. Much activity has centered around establishing the correctness of execution platforms for smartcards, and showing that applications are innocuous.

The purpose of this chapter is to motivate and illustrate applications of formal methods to increase dependability of trusted personal devices, both with respect to platform correctness and applet validation. For concreteness, we focus on devices that embed Java Virtual Machines (JVM) or their variants, in particular Java Card Virtual Machines (JCVM). Java enabled devices are a natural choice for formal methods because: i) they are widely deployed in the field; ii) they feature mechanisms that contribute to the security of the platform and the applications that execute over it; iii) detailed informal specifications of the Java platform are publicly available, and can be scrutinized. However, it should be clear that the methods presented in this paper are relevant to other execution platforms for trusted personal devices.

The remaining of this chapter is organized as follows: Section 2 begins with a brief introduction to smartcards, then continues with an overview of the JavaCard platform and a description of the software security mechanisms that it provides. Section 3 addresses the issue of platform correctness, whereas Section 4 is dedicated to application validation. We conclude in Section 5 with some perspective on emerging trends and directions for further work.

2 A Primer on Smartcards

Smartcards are a prime example of trusted personal devices in the sense that smartcards belong to a single person and are used to enable trusted operations in an information technology and communication infrastructure. Other examples of trusted personal devices include dongles for protected softwares, and under

a liberal interpretation of trusted personal devices, cell phones and other smart objects such as PDAs.

The purpose of this section is to provide a brief introduction to smartcards, starting from their characteristics and applications, pursuing with a description of the standard architecture for the current generation of smartcards, and concluding with security issues and mechanisms in smartcards.

2.1 Characteristics and Applications

Smartcards consist of a memory and a microprocessor, with special security functions, usually embedded within a credit-card sized plastic card. Depending of their type (contact or contactless cards), these cards require either a card reader, a.k.a. Card Acceptance Device or CAD for short, or a radio frequency signal for being powered. For interoperability, the standards ISO 7816 define the position of the chip on the card, the physical constraints for the connectors and the communication protocols between the chip and the reader. The CPU of the card is usually an 8 or 32-bits CISC microprocessor running at 5 MHz or more. It relies for execution on different types of memory:

- a ROM, up to 32 kB, that stores the operating system;
- an EEPROM (erasable but slow memory), up to 32 kB, that stores permanent data of the card;
- a RAM that is usually only 256 bytes in size.

The operating system on the card is responsible for communication protocols, internal memory management as well as a filesystem on the EEPROM. This filesystem organizes the memory in files and folders that correspond to the various applications loaded at the same time on the card. Accesses to the filesystem content is read/write controlled and each directory (i.e. application) can follow specific security policies so as to prevent information sharing with other applications on the card. For communication with the terminal, the operating system on the card follows the Application Protocol Data Unit (APDU) protocol. Usually, only the terminal controls the communication. However applications on the card can also use APDUs for communications with the operating system and in some cases, the card itself can initiate the communication with the terminal.

Smartcards are widely used in telecommunications (SIM cards for GSM phones and UICC cards for 3G phones), financial services (banking cards), identification (electronic ID), e-administration (digital signature), multimedia (pay-TV), transportation (contact smartcards for parkings and tolls, and contactless smartcards for public transportation) and health (electronic health records).

2.2 Open Platform Smartcards

Open platform smartcards correspond to new generation smartcards with increased flexibility. Such smartcards:

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

Java Card. Launched in 1996 by Sun, Java Card [53] is a dialect of Java adapted to the resource constraints of smartcards, and the standard programming language for smartcard applications. Java Card applets are written in a subset of the Java language, and using the Java Card API. They are then compiled down to class files, and then converted to the CAP file format; conversion involves a number of optimizations suggested by smartcard constraints, e.g. names are replaced by tokens and class files are merged together on a package basis. Finally, CAP files are loaded and installed on the card, where they can be executed by the Java Card Runtime Environment JCRE. The JCRE contains the Java Card Virtual Machine JCVM, provides support for the Java Card API and possibly for domain-specific API such as the GSM API for mobile phones, and invokes the services of a native layer to perform low-level tasks such as communication management. This Java Card architecture is summarized in Figure 1.

There are of course many similarities between Java Card and Java: in particular, Java Card programs are Java programs written in a fragment of the language, and the JCVM is a stack-based virtual machine that is closely related to the Java Virtual Machine JVM. There are also a number of differences: Java Card programs are not allowed to use large data types such as floats and strings, arrays of arrays, or finalization. Besides, some features of the JVM like dynamic class loading mechanism and multi-threading are unsupported by the

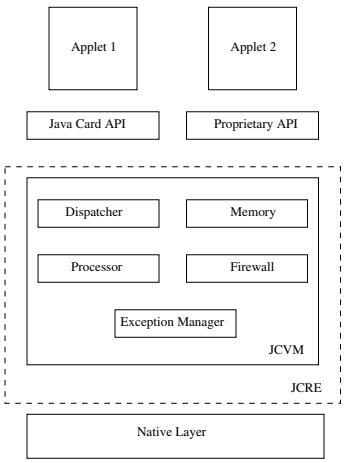


Fig. 1. Java Card architecture

JCVM (but the current version of Java Card features logical channels instead of multi-threading). Furthermore, the Java API and the Java Card API offer different functionalities, or in some cases such as remote method invocation different variants of the same functionality.

Memory management in Java Card also differs from that of Java in several aspects: in particular, garbage collection is optional and can only be performed upon completion of the execution of an applet through an explicit call to the Java Card API. Furthermore, Java Card offers a transaction mechanism for atomicity, since smart cards do not include a power supply, and thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification prescribes the use of a transaction mechanism to control synchronized updates of sensitive data. A statement block surrounded by the methods `beginTransaction` and `commitTransaction` can be considered atomic. If the transaction cannot be completed due to a card tearing or a power loss or a call to `abortTransaction`, the card will roll back its internal state to the state before the transaction was begun. If the card is unpowered, this event will occur as soon as the card is reinserted into a terminal.

Nevertheless, the essential difference from the point of view of this chapter resides in security mechanisms: indeed, Java Card abandons the Java stack inspection mechanism in favor of a simpler firewall mechanism that ensures applet isolation and that is mitigated by provisions to allow controlled communication between applets. The mechanism is discussed in the next paragraph. Further technical information about the current version of Java Card (Java Card 2.2) may be found in specifications and white papers available from Sun web site. It is likely however that Java Card will undergo a major evolution of the language and that future versions of Java Card (or successors to Java Card) will provide support for increased connectivity, and for multi-threading.

We conclude this section by mentioning that while Java Card is a central focus of this chapter, there exist other operating systems for open platform smartcards, in particular Multos, and .Net Card, as well as other dialects of Java for trusted personal devices, in particular MIDP for mobile information devices such as cell phones and PDAs, MHP for home multimedia, STIP for electronic transactions.

Global Platform. Java Card does not provide standardized mechanisms for managing applications on the card. In order to benefit from such mechanisms, most Java Cards also implement Global Platform [80], which provides a card management architecture for multi-application smartcards with post-issuance facilities (and is independent of the runtime environment).

The Global Platform architecture is detailed in two distinct specifications that respectively describe the card functional requirements and its associated security requirements. While the security of the smartcard strongly depends on a correct design and implementation of Global Platform, there has been little use of formal methods to analyze Global Platform; one notable exception is the (as yet unpublished) work of S. Zanella Béguelin, who provides a B model of

the specifications, and highlights a number of spots in the specifications where clarifications are required. The model is available from [80].

2.3 Java Card Security

The flexibility of open platform smartcards is at the same time a major asset and a major obstacle to their deployment. On the one hand, writing applets in a high-level language reduces the cost and time to market new applications, and the possibility of running several applets on a single card opens the way for novel applications. On the other hand, such smartcards introduce the possibility to load malicious applets that exploit weaknesses of the platform, make an improper use of the API, or simply launch denial-of-service attacks through immoderate resource consumption. Increased connectivity and post-issuance application loading constitute further complications from the point of view of security and contribute to making large-scale logical attacks a frightening and likely perspective.

Security Mechanisms. Current security architectures for smartcards feature two central mechanisms, a.k.a. security functions, to prevent logical attacks: the firewall, and the bytecode verifier.

Firewall. In the Java Card environment, the security model for partition between applications and between operating system and applications differs from the sandboxing model of Java. In the firewall model of Java Card, an unique context (Applet IDentifier or AID) is associated to each applet on the card. Only one context can be active, the one from the current applet. The JCRE prevents any access from one object to another object with another context, with the exceptions of static variables, arrays declared as global or instances declared as entry points. In the other cases, objects must communicate exchanging objects implementing the `javacard.framework.Shareable` interface. This procedure remains controlled by the JCRE and is the following:

- The server applet must define an interface `SI` extending the `Shareable` interface, a class `C` implementing `SI` and create an object `O` of class `C`;
- To access the object `O` from applet `A`, the client applet `B` must invoke the `JCSystem.getAppletShareableInterface` method;
- The JCRE uses the `getShareableInterfaceObject` method to send a request to Applet `A`. Then Applet `A` determines, given the AID of `B`, if `B` is authorized to access `O` and if so, returns a reference to `O`;
- The applet `B` casts `O` to type `SI` into a class reference `SIO`;
- The firewall prevents the applet `B` from accessing any field or method not defined in `SI`.

An example of such a procedure is given in Figure 2, where the applet `Bob` want to share an object belonging to the applet `Alice`:

```
public interface SI extends Shareable {
  Secret foo(); }

public class Alice extends Applet implements SI {
  private Secret ObjectSecret;

  public Shareable getShareableInterfaceObject(AID Client) {
    if (Client.equals(BobAID))
      return(this);
    return null; }

  public Secret foo() {
    AID Client;
    Secret Response;
    Client = JCSystem.getPreviousContextAID();
    if (Client.equals(BobAID))
      Response = ObjectSecret;
    return Response; } }

public class Bob extends Applet {
  public static SI AliceObj;
  private static Secret AliceSecret;

  public void bar() {
    AliceObj = (SI) JCSystem.getAppletShareableInterface
                      (AliceAID);
    AliceSecret = AliceObj.foo(); } }
```

Fig. 2. Example of the use a `Shareable` object

Bytecode Verification

Goals. The bytecode verifier is a key security function in the Java Card architecture. Its purpose is to check that applets are correctly formed and correctly typed, and that they do not attempt to perform malicious operations during their execution. It consists on a two steps process. The first one, and the simplest, is a structural analysis of the consistency of the CAP file and its constant pool. The second one requires a static analysis of the program and is meant to ensure that:

- values are used with their correct type (to avoid forged pointers) and method signatures are respected;
- no frame stack or operand stack underflow or overflow will occur;
- visibility of methods (**`private`**, **`public`**, or **`protected`**) is compatible with their use;
- objects and local variables are initialized before being accessed. Together with subroutines, which are discussed below, initialization is one of the main

difficulties from the point of view of bytecode verification, as illustrated e.g. by S. Freund and J. Mitchell [47];
– jumps in the program code remain in legal bounds.

Ensuring such properties is an important step towards guaranteeing security, and the failure to enforce any of these properties may be exploited for launching attacks.

Algorithms. Bytecode verification [63] is a data-flow analysis of a typed virtual machine which operates on the same principles that the standard JVM except for two crucial differences: the typed virtual machine manipulates types instead of values, and executes one method at the time.

The data-flow analysis aims at computing solutions of data-flow equations over a lattice derived from the subtyping relation between JVM types, and uses to this end a generic algorithm due to G. Kildall [57]. In a nutshell, the algorithm manipulates so-called stackmaps that store for each program point an history structure that represents the program states that have been previously reached at this program point. The history structure is initialized to the initial state of the method being verified for the first program point, and to a default state for the other program points. One step of execution proceeds by iterating the execution function of the virtual machine over the states of the history structure. Each non-default state is chosen once and the result of the execution of the typed virtual machine on this state is propagated to its possible successors in the history structure.

Different history structures can be used depending on the accuracy required from the analysis.

– In a monovariant analysis, the history structure stores one program state, which is the least upper bound of the states that have been been previously computed at this program point. In such an analysis, propagating a state in an history structure amounts to taking pointwise the least upper bound (on the type lattice of the virtual machine) of the types appearing in the two states and storing the result back at this location. The termination of the analysis is guaranteed since the set of states does not have an infinite ascending chain, and the state stored in the history structure is increasing. As noted by R. Stata and M. Abadi [93], collapsing history structures to a single state as done in the monovariant analysis leads to a bytecode verification algorithm that does not handle subroutines as prescribed by the informal specifications of Sun. To be more precise, monovariant bytecode verification rejects bytecode programs that make a polymorphic use of subroutines. This use of subroutines can lead to two states, for a same program point, that do not have the same number of local variables or the same number of elements in the operand stack and that would then be merged state into an error state, although the execution is valid.
– In a polyvariant analysis, the history structure stores the set of program states that have been previously computed at this program point. In such an analysis, propagating a state in an history structure amounts to adding

the newly computed state to the history structure. The termination of the analysis is guaranteed since the set of states is finite, and the size of the history structure is increasing.
Polyvariant bytecode verification provides an accurate treatment of subroutines, and was introduced independently by P. Brisset (in unpublished work) and by A. Coglio [27]. L. Henrio and B. Serpette [49] propose an improvement of polyvariant bytecode verification in which compatible states in the history structure can be merged so as to keep the size of history structures reasonable. It is interesting that approaching bytecode verification through model-checking [81,10] results in an analysis which capture a similar class of programs as polyvariant bytecode verification.

We refer the reader to [63] for a more detailed account of algorithms for bytecode verification.

On-Device Verification. Currently applets are verified off-card and, in case of a successful verification, signed and loaded on-card. Such a solution is not optimal in the sense that it leaves a crucial component of the security architecture outside of the perimeter of the smartcard. However, there are several proposals for circumscribing the trusted computing base to the smartcard using on-card bytecode verification. One solution adopted in the KVM [29] is to rely on lightweight bytecode verification, initially proposed by E. Rose, in which the program comes equipped with the solution to the dataflow equations, and the role of the verifier is to check that the solutions are correct. Another proposal by X. Leroy [62] is to perform an off-card transformation that allows bytecode verification to be performed in one pass. A later work by D. Deville and G. Grimaud [34] does not require programs to be rewritten or annotated, but exploits instead efficient encodings of the data structures manipulated by the bytecode verifier.

Security Issues. The Java Card security architecture guarantees that downloaded applications are innocuous and comply with some basic policies related to typing, initialization or access control. Such basic policies are the cornerstones upon which the overall security of the smartcard will rely. Therefore it is important to verify that the security architecture does enforce these basic policies as intended. Thus, an important application of formal methods to smartcard security is platform verification, which aims at providing an abstract model of the Java Card platform and security architecture, and at proving that the security functions play their expected role. However, it is not sufficient to show that security functions are correctly designed. In particular, one also has to ensure that other components of the infrastructure are correctly designed: the Java Card API and the Global Platform API constitute two prominent components of the infrastructure whose correct design is central to security. Thus, another important aspect of platform verification is to show the API are correctly designed.

Platform verification is a fundamental step towards guaranteeing the security of smartcards, and a prerequisite for Common Criteria evaluations at the

highest levels. Nevertheless, the guarantees offered by the Java Card security architecture are limited, and further verifications must be performed to verify that applications make a legitimate use of the infrastructure, and do not attempt any hostile action. Thus, application validation is another important application of formal methods to smartcard security. There are many facets to applet validation, each with its own objectives and techniques. For example, applet validation may be performed at bytecode level prior to loading an application on card. Another scenario is that applet validation is performed at source level by developers or experts in formal methods working with developers (this is an ideal situation, often formal methods are used at posteriori). While such a scenario is not ideal from the perspective of guaranteeing the security of a smartcard, the smartcard industry has found such a scenario particularly useful in particular for checking that applets respect some given security requirements.

In summary, security is a holistic property of a system, and formal methods must therefore be employed at many different levels to provide strong guarantees about smartcard correctness. Platform verification and application validation are two important aspects of guaranteeing security for smartcards, and the focus of the next sections. Other important aspects of formal methods which are not treated in this chapter include the use of formal methods to establish a relation between the models developed for platform verification and the actual implementations of the platform, see e.g. [23], or the use of formal methods to verify cryptographic algorithms or protocols used by smartcards, see e.g. [67].

3 Platform Certification

Dedicated operating systems for smartcards aims at providing a secure environment for applications execution. For this purpose, special security features are provided, and precise specifications on the platform are given. However, due to the size of these specifications, the use of formal methods is required to ensure to the whole specifications and the corresponding implementation are correct. For instance, once the entire has been formalized, it is possible to give the statement of global properties, such as type correctness or applets isolation, and prove that the platform do not contain design flaws or implementation bugs for these properties. Although the approach was different at the time, a type-system related security hole was indeed found in the bytecode verifier of Java 1.1 by the Kimera project.

Several formalizations of the Java (Card) platform are now available. They differ by the coverage of the runtime environment and virtual machine, the formalism they are built upon, the style of semantics used for the instruction set and the particular aspect of security they aim at verifying. The Isabelle/HOL formalizations of T. Nipkow and co-workers [58] and the executable specifications of the J-book [92] constitute some of the most impressive achievements in this direction to date. The reader may refer to [48] for a not so up-to-date survey of the various formalizations. In the following, we will focus on a another formalization [38] not yet available at the time of the survey. This formalization,

written within the Coq proof assistant, covers almost all the aspects of the Java Card platform, has not be written for a specific security property and thus remains general purpose. Besides, it is executable (we consider executability as an essential point to remain as close as possible of a reference virtual machine) and it comes with a tool to resolve constant pool of CAP files and translate them, including the ones with native methods, into the representation of programs given below.

3.1 Formalisms

The following formalization of the Java Card platform has been written in the Coq [30] specification language. Nevertheless, the formalization is easily translatable to other programming language such as CAML, and other proof assistants such as Isabelle and PVS, since it does not use any high level feature of Coq. We will describe in the following the corresponding subset of the specification language.

The keyword **Inductive** (resp. **Mutual Inductive**) introduces a inductive (resp. mutual inductive) type. Such definitions also declare all constructors for these types. We give below the definitions of the natural numbers `nat` (with the constructors `O` and `succ`, for successor), of parameterized (polymorphic) lists `list`, and of the parameterized `option` type commonly used to formalize non total functions (lift monad):

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.

Inductive list (A:Set) : Set :=
| Nil : list A
| Cons : A → list A → list A.

Inductive option (A:Set) : Set :=
| None : option A
| Some : A → option A.
```

The partial functions `head`, `tail` respectively return for a given list the first element, and the list without its first element. Our formalization uses the type `list` in many places as the type of ordered sets, for instances to represent stacks or arrays. This allows to use all predefined accessors functions and lemmas of Coq and to get a directly executable semantics. However, with the new module system of Coq, it would also be possible to declare abstract modules for basic datatypes and give later an executable implementation of these modules.

Records are introduced by keyword **Record**. Accessors of these fields are then expressed as functions named by the corresponding fields of the record. The construction `match...with...end` introduces pattern matching as in ML languages.

3.2 Virtual Machine

The virtual machine described in the following is a defensive virtual machine, i.e. it follows the specification of the reference virtual machine by explicitly implementing all the pre-conditions (the *must* and *must not* clauses) stated by the official specification. This virtual machine is based on Java Card version 2.1, supports the full instruction set of the specification, includes the firewall, but lacks some other features of the Runtime Environment that rely on native APIs (such as APDUs or transactions), although a methodology to include these APIs is given.

Java Card Programs. In this section, we outline the Coq formalization of Java Card bytecode programs, as executed by the virtual machine. The reader may refer to [38] for a complete description of this formalization.

Type system. The Java Card Virtual Machine distinguishes between primitive types and reference types. Each of these types is subdivided into atomic types. For instance, there are four kinds of reference types that are null types, array types, instance types and interface types. Arrays are described according to the type of their elements which must be primitive or reference. The reference of an instance or an interface corresponds to an index in the set of classes or interfaces of the program. Finally, the type of Java Card types is given by the following mutual inductive definition that captures the considerations below:

```
Inductive vmtype : Set :=
  | Prim : vmtype_prim → vmtype
  | Ref : vmtype_ref → vmtype
with vmtype_ref : Set :=
  | Ref_null : vmtype_ref
  | Ref_array : vmtype → vmtype_ref
  | Ref_instance : class_idx → vmtype_ref
  | Ref_interface : interf_idx → vmtype_ref.
```

where `vmtype_prim` is a simple inductive type that gathers all primitive JCVM types.

We notice that the constructor for arrays, `Ref_array` allows to form types corresponding to arrays of arrays, which is not permitted in Java Card. However our formalization of the operational semantics of the JCVM, and in particular the implementation of the `anewarray` instruction, does not allow to form such a type.

Programs. Our virtual machine does not work directly on the binary representation of the CAP file format but on structured representation of programs, obtained by a dedicated tool. After this transformation, that resolves the constant pool of the program, only the essential components of a typed object-oriented language remain: types, methods, classes and interfaces. Formally, a program is described in Coq by the following record type:

```
Record jcprogram := {
```

```
  interfaces : (list Interface);
  classes    : (list Class);
  methods    : (list Method);
  sheap_type : (list type)
}.
```

where the types `Interface`, `Class`, `Method` and `type` are themselves defined as record types. `sheap_type` is used for initialization purposes, to determine types of variables declared as static in the program. For simplicity, we only deal with closed programs hence the packages `java.lang` and `javacard.framework` are an integral part of programs.

Methods. A method is characterized by its status (static or not), its signature (against which one can type-check its arguments upon invocation and returned value upon completion of the method), the number of its local variables (for initializing its execution context), its list of instructions to be executed (the bytecode), its exception handlers (a handler is an object that identifies the code for managing dynamic errors), its maximum operand stack size and finally its owning class and the indexes of the method. Formally, we use the following structure to represent methods:

```
Record Method : Set := {
  is_static          : bool;
  signature          : signature_type;
  nb_local           : nat;
  bytecode           : (list Instruction);
  handlers           : (list handler_type);
  m_max_opstack_size : nat;
  owner              : class_idx;
  method_id          : method_idx
}.
```

where `class_idx` and `method_idx` corresponds respectively to the type of indexes to a class and to a method. Others components of a program, such as interfaces and classes, are represented in a similar fashion.

The type `Instruction` enumerates with an inductive type the different bytecodes of the Java Card language, with theirs operands:

```
Inductive Instruction : Set :=
  | nop           : Instruction
  | push          : vmtype_prim → Z → Instruction
  | ret           : locvars_idx → Instruction
  | invokespecial : nat → method_idx → Instruction
  | invokestatic  : nat → method_idx → Instruction
  | getfield      : vmtype → instance_field_idx → Instruction
  | inc           : vmtype_prim → Z → nat → Instruction
...
```

where `Z` is the type of binary integers. Some bytecodes with similar semantics have been collapsed into a single one. For instance, the bytecode `inc` takes

as a supplementary argument a primitive type and thus represents the bytecodes `iinc` et `sinc` of Java Card (incrementation of a register of type `Int` and `Short` respectively). It is possible to represent the 185 Java Card bytecodes by only 44 bytecodes in our formalization. Also, for convenience, slight differences may appear in the operands of instructions, such as for `invokestatic` and `invokespecial` that receive an extra parameter corresponding to the number of arguments of the method.

Each instruction of the virtual machine is given by a small-step semantics; more precisely, each instruction is formalized as a state transformer, i.e. a function that takes as input a state (before the instruction is executed) and returns a new state (after the instruction has been executed).

Memory Model. Java (Card) virtual machine is stack-based. For intermediary computations, values are pushed and popped from an operand stack. Also, execution contexts for methods, called frames, are organized in stack, each new invoked method being pushed in top of the stack, and thus becoming the active frame. In our formalization, the state contains all the dynamic items manipulated by a Java Card program: values, an heap (for created objects) and a stack of frames.

Values. Values constitute the main element manipulated by the virtual machine. In a defensive virtual machine, values are typed and follow a definition similar to the type system, with an inductive type, where we distinguish primitive and reference values. Type information is carried by the constructor and the numerical value by the argument of the constructor. We obtain for reference values:

```
Inductive d_val_prim :=
  | d_ReturnAddress : bytecode_idx → d_val_prim
  | d_Void          : Z → d_val_prim
  | d_Boolean       : Z → d_val_prim
  | d_Byte          : Z → d_val_prim
  | d_Short         : Z → d_val_prim
  | d_Int           : Z → d_val_prim.

Inductive d_val_ref : Set :=
  | d_Ref_null : d_val_ref
  | d_Ref_array : vmtype → heap_idx → d_val_ref
  | d_Ref_instance : class_idx → heap_idx → d_val_ref.
```

where `heap_idx` indicates the location of an object in memory and `class_idx` is an index into the program classes. There is no value corresponding to interfaces, since an interface must be implemented by a class to be used, through an instance.

Finally, the Coq type of Java Card values is defined by:

```
Inductive d_val : Set :=
  | d_Prim : d_val_prim → d_val
  | d_Ref : d_val_ref → d_val
  | d_NonInit : class_idx → bytecode_idx → heap_idx → d_val.
```

where `d_NonInit` is a specific value of non-initialized class instances.

Objects. Objects in memory constitutes the runtime representation of classes and arrays. The `d_Ref_instance` and `d_Ref_array` values defined in the previous section refer to these objects. We collect in a record information needed for each of these types.

We define an inductive type for objects:

```
Inductive obj : Set :=
  | Instance : type_instance → obj
  | Array    : type_array → obj.
```

where `type_instance` and `type_array` are record types containing all dynamic information related to the object, such as values of the fields of instances or values of arrays. Finally, the memory area in which objects are stored during the execution of a program, the heap, is naturally defined as:

```
Definition heap := (list obj).
```

Frames and Stack. Frames contains computational information for methods: an operand stack and a set of local variables (or registers). An index records the referring method, and a program counter points to the next instruction to execute within the method. Finally, the maximum operand stack size is duplicated within the frame to avoid frequent lookup in the method referenced in `method_loc`.

```
Record d_frame : Set := {
  d_opstack          : (list d_val);
  d_locvars          : (list (option d_val));
  d_method_loc       : method_idx;
  d_p_count          : bytecode_idx;
  d_context_ref      : AID;
  d_max_opstack_size : nat
}.
```

where `AID` is the type for an unique identifier of applets (for the firewall mechanism). The type `option` of the field `locvars` will be used to formalize non initialized local variables.

States. The notion of states is defined by the following record:

```
Record d_state : Set := {
  d_sfields_f : d_sfields;
  d_heap_f  : d_heap;
  d_stack_f : (stack d_frame)
}.
```

where `d_sfields` is the type for the static fields (a list of `d_val`, in which will be stored variables declared as static and shared by the whole program). Most of the instructions will act on the top frame, i.e. the head of the stack. Other fields might be added to the notion of states such as input/output buffers for APDUs (see Section 2.1).

Finally, states are used for the construction of return states of instructions. Instruction can progress normally, provoke an error or throw an exception:

```
Inductive d_rstate : Set :=
  | d_Normal         : d_state → d_rstate
  | d_Abnormal       : eLabel → d_state → d_rstate
  | d_ThrowException : xLabel → d_state → d_rstate.
```

where `eLabel` and `xLabel` are inductive types that record the reason of the error or the exception respectively.

Semantics of Instructions. Bytecode instructions update the memory of the JCVM according to operands generated at compile time and to the representation of programs given below. Thus, the main execution function of the virtual machine, that determines which is the next instruction to execute and calls the corresponding function in our formalization, has the following signature:
`d_exec : d_state →d_rstate`
Then, most instructions have a similar execution pattern:

1. the initial state is decomposed;
2. fragments of the state are retrieved;
3. observations are made to determine the new state;
4. the final state is built on the basis of the retrieved fragments and of the observations made.

We illustrate this pattern on the bytecode `ifnull` that compares the first element of the operand stack, that must be a reference value, to null and branches accordingly to the program counter given as a parameter or to the next instruction. The defensive semantics of this bytecode (defensive in the sense it enforces type verification as mentioned in Sun specification and may return a `type_error`) is given by the following `d_ifnull` Coq function:

```
Definition d_ifnull (b : bytecode_idx) (s : d_state) :=
  match d_stack_f s with
  | Nil ⇒ d_Abnormal state_error
  | Cons h lf ⇒
      match head (d_opstack h) with
      | Some v ⇒
          match v with
          | (d_Ref _) ⇒
              match d_res_null v with
              | True ⇒
                 d_update_frame (d_update_pc b
                                   (d_update_opstack
                                     (tail (d_opstack h)) h)) s
              | False ⇒
                 d_update_frame (d_update_pc (succ (d_pc h))
                                   (d_update_opstack
                                     (tail (d_opstack h)) h)) s
              end
          | _ ⇒ d_Abnormal type_error
          end
      | None ⇒ d_Abnormal opstack_error
      end
  end.
```

where the d_res_null checks for null pointers and d_update_frame function takes as a parameter the current updated frame h, a state s and returns a normal rstate which is an update of s where the topmost frame of the execution stack has been replaced by h.

Other security checks are illustrated within the following excerpts of the invokevirtual semantics. The function new_frame_invokevirtual is used once the initial state has been decomposed. Formally, we set:

```
Definition new_frame_invokevirtual (nargs : nat) (m : Method)
 (nhp : obj) (h : d_frame) (st : d_state) (cap : jcprogram) :=
 (* Extraction and removal of the arguments *)
match l_take nargs (d_opstack h), l_drop nargs (d_opstack h)with
  | Some l, Some l' ⇒
    (* Security check *)
    if test_security_invokevirtual h nhp
    then
      (* Signature check *)
        if (signature_verification l (signature m) cap)
        (* Updates the current frame and pushes the new frame *)
        then d_Normal ...
        else d_AbortCode signature_error st
    else d_ThrowException Security st
  | _, _ ⇒ d_AbortCode opstack_error st
  end.
```

where h is the current topmost frame, obj is the object on which the method m should be invoked and the ellipsis stands for the resulting built state (omitted in the excerpt). The function performs various checks, such as the verification of the arguments of the method w.r.t its signature and the firewall mechanism with test_security_invokevirtual, that may throws a SecurityException. For example, in case the object nhp is an instance, the function will verify whether (1) the active context is the Java Card Runtime Environment context or; (2) the active context is also the context of the instance owner or; (3) the instance is an entry point. If not, the function returns true to flag a security violation. Formally:

```
Definition test_security_invokevirtual (h : d_frame)(nhp: obj):=
  (* Tests if the active context is the JCRE *)
  if eqb_AID (context_ref h) jcre_AID
  then true
  else
   match nhp with
     (* The object is an instance *)
     | Instance ti ⇒
       (* Checks for equal contexts or entry point *)
       orb (eqb_AID (context_ref h) (owner_i ti))  (ptE ti)

     (* The object is an array *)
     | Array ta ⇒
       (* Checks for equal contexts or global array *)
```

```
      if eqb_AID (context_ref h) (owner_a ta)
      then true
      else eqb (statusglobal ta) is_global
end.
```

3.3 Javacard API

In addition to the Java Card Virtual Machine, the Java Card Runtime Environment includes an implementation of the Java Card APIs. This implementation is required to execute those Java Card programs that appeal to the APIs.

In order to obtain a complete Java Card Runtime Environment and to be able to execute or reason about any Java Card program, we must therefore formalize the APIs in Coq. For the APIs that rely on standard Java Card code, the modeling is direct since we can represent them as any Java Card program.

However, special care is required to deal with the APIs that deal with native methods whose code is not written in Java. For such methods, we need to provide an implementation in Coq. In our formalization, we give the implementation of some native methods from the APIs, such as the method `arrayCopy` which appears in the example of Section 4.2 and is used to copy an array. Examples of APIs that are not treated that are not treated by our formalization include for instance the APIs for communication with APDUs, required for instance to select and execute a particular applet. Formalizing these APDUs could be achieved by adding into our definition of a state two fields corresponding to an input and an output buffer [91].

With the complete implementation of the APIs, the starting point of our formalization for executing a program could then be the `main` method from the `Dispatcher` class of the `javacard.framework` package, i.e. the standard starting point of a Java Card smart card after any reset.

3.4 Verified Java(Card) Bytecode Verifiers

In this section, we present the general methodology that we used for verifying bytecode verifiers [7]. In addition to the defensive virtual machine (from Section 3.2), the methodology involves a typed virtual machine, and an offensive virtual machine.

Typed and Offensive Virtual Machine. The bytecode verifier is built on a variant of the defensive virtual machine, called typed (abstract) virtual machine, that uses types as values and thus only performs type verifications. Corresponding datatypes for values are given below (and prefixed with `a_`) and extended to the notion of typed states `a_state` and typed return states `a_rstate`:

```
Inductive a_val_prim :=
| a_ReturnAddress : bytecode_idx → a_val_prim
| a_Void          : a_val_prim
| a_Boolean       : a_val_prim
| a_Byte          : a_val_prim
| a_Short         : a_val_prim
| a_Int           : a_val_prim.
```

```
Mutual Inductive a_val :=
| a_Prim          : a_val_prim → a_val
| a_Ref           : a_val_ref → a_val
with a_val_ref :=
| a_Ref_null      : a_val_ref
| a_Ref_array     : vmtype → a_val_ref
| a_Ref_instance  : class_idx → a_val_ref
| a_Ref_interface : interf_idx → a_val_ref.
```

Numerical values of the defensive values have been removed, except for the numerical value of return addresses which is a static information and is used to determine the control flow for subroutines instructions. The semantics of bytecode is modified according to these datatypes, however for some bytecodes (branching instructions for instance), this virtual machine is non-deterministic due to the loss of computational information. Possible resulting states are collected together as the result of bytecode execution, and then, the main execution function has the following signature:

```
a_exec : a_state →(set a_rstate)
```

The correctness of this virtual machine w.r.t to the defensive virtual machine is expressed through cross-validation. Given the obvious abstraction functions `alpha_da` (resp. `alpha_da_rs`) from defensive state to typed state (resp. return states), the diagram from Figure 3 relating defensive execution `d_exec` and typed execution `a_exec` must be commuting. Note that in this Figure, the curved arrow to `set a_rstate` denotes set inclusion, and that this diagram excludes invocation and return instructions that are handled differently (see [5] for more details).

The offensive virtual machine is built on a similar basis, but uses untyped values and do not perform type verification.

```
Definition o_val_prim := Z.
Definition o_val_ref := Z.
Definition o_val := Z.
```

This virtual machine is closer to a real implementation (faster than the defensive virtual machine for execution, since it does not perform type verification). It has also be proved as safe as the defensive virtual machine, provided the bytecode verification of the executed program has been successful.

The tool Jakarta [5] has been design to generate, given abstraction functions from one virtual machine to another, the abstracted virtual machine as well as proofs of cross-validation. The abstraction process is guided by a script for com-

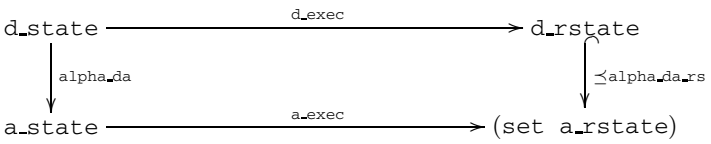


Fig. 3. Commutative diagram of defensive and typed execution

plex cases, however only 150 lines of script are needed to produce the expected results for the typed virtual machine from the 5,000 lines long defensive virtual machine. The offensive virtual machine can also be obtain with Jakarta and cross validation results follow a commuting diagram similar to the typed virtual machine. However, for this diagram, the commutation is limited to the cases where the defensive execution does not lead to a type error, since the offensive virtual machine assumes that such errors do not happen. It can also be written with the following formula:

```
∀ (s:d_state), d_exec s ≠ d_Abnormal type_error →
                alpha_do_rs (d_exec s) = o_exec (alpha_do s)
```

where `alpha_do`, `alpha_do_rs` and `o_exec` are similar to `alpha_da`, `alpha_da_rs` and `a_exec` but for the offensive virtual machine.

Abstract Definition and Construction. The formalization of the bytecode verifier relies on the modules system of Coq. It offers a refined model of the various notions (transition system, fixpoint structure, bytecode verifier, abstract virtual machine, etc.) involved to obtain the bytecode verifier for the defensive virtual machine.

A *bytecode verifier* is given by a type `state` of *states*, an *execution relation* `exec` over states, a set `err` of error states and a predicate `check` such as:

```
forall a:state, (check a) → ¬(bad a).
```

where a state is `bad`, if it is possible to reach from it an error state by successive transitions of the execution relation. Thus the predicate `check` rejects all states that lead by execution to an error state.

The standard way to build such a bytecode verifier is to endow the type of states with a order that does not admit infinite ascending chains, and for which execution is increasing (to guarantee termination), and such that error states are upwards closed. If furthermore execution is deterministic, one can compute for every state `a`, the least fixpoint `b` upper `a`. To do so, we define for every state `a` the least fixpoint `lfp a` below it as:

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

Then, we define `check a` as `(err (gfp a))`. As execution is monotone and `lfp a` is the least fixpoint upper `a`, it is clear that such a checking is sufficient to guarantee that `a` is not a bad state.

In a first step, the function `exec` from the above construction is instantiated to the execution function from the typed virtual machine running over the corresponding stackmap for the chosen analysis (monovariant, polyvariant described in Section 2.3). This leads to a verification based on a single method for the typed execution. However, this result can be extended to a result about the defensive execution, by verifying individually each method of the program

being verified, by using cross-validation results between typed and defensive virtual machines, and by appealing to a complex invariant between the typed and defensive virtual machine for the case of instructions that change the current frame such as invocations, returns or instructions that raise an exception. Finally, using this bytecode verifier for the defensive virtual machine and results of cross-validation between defensive and offensive virtual machines (including some extras properties on method invocation), we can easily obtain the following expected property:

```
forall s:d_state, (check s) →
(alpha_do_rs (d_exec s)) = (o_exec s).
```

If `s` is the initial state of a program, this property does guarantee that if the verification of the program has been successful, than defensive and offensive execution coincide. This is captured formally by introducing functions that perform several steps of execution, and by showing

```
forall s:d_state, forall n:nat, (check s) →
(alpha_do_rs (d_exec+ n s)) = (o_exec+ n s).
```

As summarized in Figure 4, to take advantage of the bytecode verification framework, the user must provide:

- a defensive virtual machine;
- the definition of abstraction functions for Jakarta abstraction scripts, that are used to construct the abstract virtual machine and an offensive virtual machine. Scripts may contain some minimal amount of proof information to carry cross-validation;
- a formal proof of the correctness w.r.t. bytecode verification of method invocation and exception handling;
- a choice of an analysis and of the corresponding history structures.

Then the user obtains 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.

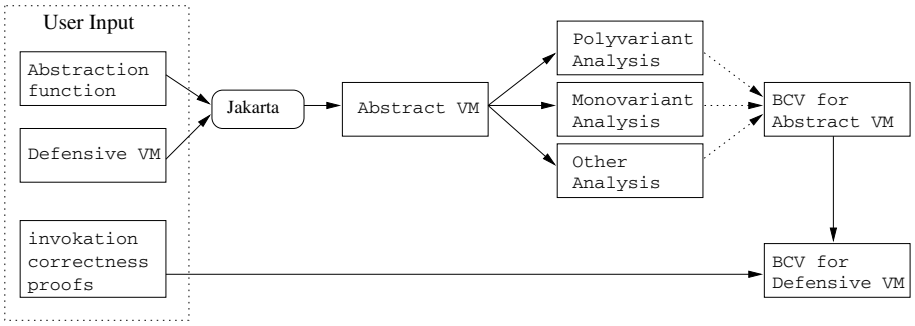


Fig. 4. Construction of verified bytecode verifier

4 Application Validation

While the correctness of the runtime environment is an essential to guarantee the overall security of trusted personal devices, no device can be deemed secure without ensuring that applications do not behave maliciously, in particular with respect to API usage.

The validation of applications against security policies can be addressed at different levels:

- one can enhance existing security architectures to enforce security properties not addressed by current architectures, in particular confidentiality and availability. Verification can be performed by enhanced bytecode verification mechanisms;
- one can abandon the realm of type systems and its associated benefits and choose develop logical methods for specifying and verifying either automatically or efficiently a specific class of security properties. Verification can be performed by (possibly efficient and hence incomplete) logic-based proof inference mechanisms;
- one can exploit the expressive power of logical methods to require that applications, or at least sensitive fragments of applications, are subjected to functional verification, i.e. to verifications that establish their correctness in terms of functionality as well as security.

The different levels may be combined, e.g. by using alias type systems to improve the modularity of functional verification [72]. For the clarity of presentation, we never introduce these different levels separately. In Section 4.1, we discuss possible applications of static analyses and type systems for guaranteeing the security of applications. In Section 4.2, we consider logical verification techniques for verifying specific security properties as well as functional correctness. The paragraph takes the point of view of code developers who want to increase their confidence in the code they develop, and therefore focuses on source program verification; nevertheless we briefly discuss the point of view of the code consumer at the end of this section.

4.1 Enhanced Type Systems and Static Analyses

Program analysis techniques such as type systems and static analysis provide a well-established means to enforce program properties at compile-time or load-time. In the domain of Java-enabled devices, the prime example of program analysis based on type systems is bytecode verification, which is discussed in Section 3.4. In addition, there are several proposals of type systems and static analysis for JVM programs, in particular for eliminating dynamic checks that incur a loss of performance at execution time, and for enforcing security properties that are not guaranteed by the Java security mechanisms.

Static Analyses for Access Control. While the runtime penalty incurred by dynamic access control is acceptable in practice, it is desirable to detect

statically that an application may attempt to violate the rules of the firewall, since such attempts will result in a security exception that may block the card. D. Caromel, L. Henrio and B. Serpette [22] have proposed a static analysis that detects statically whether an application may raise a security exception, whereas M. Eluard and T. Jensen [40] have proposed a similar but finer analysis that covers more precise sharing policies. W. Dietl, P. Müller and A. Poetzsch-Heffter [36] manage a similar effect using a type system adapted from ownership type systems. In order for their approach to be practical, downcasts allow to add information to references that can belong to any context into more specific references, e.g. the reference belongs to the currently active context, or is an entry point (of course, such additional information must be verified). The information provided by downcasts is used in type checking, thus the correctness of the type system can only be achieved by combining type checking with runtime checks that verify at that the information associated to downcasts is correct.

An example of a rejected program for these analysis can be built from the program of Figure 2: Suppose that the applet `Alice` contains a public method `baz`, then inside the method `bar` from applet `Bob` a call to `AliceObj.baz()` would lead to a security exception since this `Alice` and `Bob` belong to different contexts and the invoked method does not belong to the shareable interface `SI`.

In a series of articles, T. Jensen and his co-workers [13,14] propose a method to check control-flow properties of Java applets. Their method relies on constructing a finite-state automaton that approximates of the control-flow graph of an application, and checking the automaton against security properties expressed in temporal logic. The method is applicable to stack inspection as well as other properties. Building up on earlier work by F. Pottier, C. Skalka and S. Smith [82], T. Higushi and A. Ohori [50] propose a static type system for access control in the JVM, and develop a sound inference algorithm to verify statically that programs respect the access control policy specified by the type system.

Static Analyses for Secure Information Flow. The firewall mechanism provides a means to control which principals access but does not guarantee that confidential information will not leak to unauthorized principals [71]. In order to avoid principals (that may access information legitimately) to pass the information unduly, it is desirable to devise security mechanisms that enforce stronger confidentiality policies such as non-interference, a high-level security property that guarantees the absence of illicit information flows during a program execution. Non-interference assumes that the variables in a program are either public (low) or secret (high), and requires that the initial values of secret variables do not influence the final values of public variables.

A. Myers [73] and A. Banerjee and D. Naumann [2] propose static enforcement mechanisms for guaranteeing non-interference of Java programs, using extended programs in which methods are declared with security signatures and fields are declared with their security level (in fact, [2] combines information flow and access control but we gloss over the issue here). A example of class declaration in the language of [2] is:

```
class PatientRecord extends Object {
name : (string, L);
hivstatus : (bool, H);
drug : (string, H);

(void, L) setDrug >-H→ (string, L) {
  ...
  }
...
}
```

where the text `>-H->` in the declaration of the method `setDrug` indicates that the method has a high heap effect, i.e. will modify high parts of the heap that store the values of high fields. While the type system of [2] has been proved sound in the sense that typable programs are non-interfering, the type system of [73], which exhibits a richer set of features, lacks a proof of soundness.

An information flow type system for a representative fragment of the JVM that includes classes, methods, and exceptions is given in [9]; the type system is compatible with bytecode verification, and is sound, in that it rejects non-interfering programs. The examples below illustrate some of the illicit information flows that can occur in a stack-based language, and that are detected by the type system of [9]:

1 sload y_H	1 spush 3
2 if_eq 6	2 spush 4
3 spush 0	3 sload y_H
4 sstore x_L	4 if_eq 6
5 goto 8	5 sstore y_H
6 spush 1	6 sstore x_L
7 sstore x_L	7 return
8 return	

These programs are example of indirect flows, since the final value of the low variable x_L depends on the initial value of the high variable y_H. The problem is caused in the first case by an assignment to x_L in the scope of a branching instruction, and in the second case by an instruction that manipulates the operand stack in the scope of a branching instruction.

To prevent such illicit flows, the type system of [9] manipulates typed states that include security environments, i.e. maps that assign a security level to each program point, and allows branching instructions to update the security environment, e.g. if they branch over a high value.

Although the prevention of illicit information flows is an important concern for the smartcard industry [15,66], and although there have been important achievements in the design of static enforcement methods for information flow security policies [87], the methods have not found substantial applications in practice, partly because information flow policies based on non-interference are too rigid and do not authorize information release, whereas security sensitive applications often release deliberately some amount of sensitive information.

Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases.

In a recent survey [88], A. Sabelfeld and D. Sands provide an overview of relaxed policies that allow for some amount of information release, and a classification along several dimensions, for example who releases the information, and what information is released. While several information flow type systems have been developed to accommodate some dimensions of information release, it is likely that providing flexible static enforcement mechanisms that integrate the different dimensions of declassification will contribute significantly towards a wider use of information flow type systems.

Other Static Analyses and Type Systems. The paragraphs above illustrate some prominent applications of type systems for enforcing security of JVM applications but the scope of properties enforceable by type systems is much wider: for example, G. Schneider *et al* [21,90] have recently developed an efficient analysis to estimate memory usage for Java smartcards. Advanced type systems can also be used to enforce safety policies or to justify aggressive optimization strategies. Existing analyses for JVM programs include array-out-of-bounds analysis [97] using restricted dependent types [96], exception analysis [56] and escape analysis [16], as well as type systems for concurrent fragments of the JVM [41,60].

Implementations. It is noticeable that advanced type systems for the JVM have remained at the level of prototype implementations, and have not found their way in security architectures. While the situation is partially a natural consequence of the exploratory nature of some type systems, P. Fong [44] suggests that the situation may also result from a more fundamental limitation of the Java platform, namely that the verification architecture of the JVM is not designed to support extensibility, and advocates the design of extensible protection mechanisms that can be used to accommodate mechanisms tailored towards enforcing application-specific properties such as confidentiality, access control, or resource management. Building on earlier work on proof linking [45], he proposes an architecture that supports pluggable verification modules, and illustrates the principles of his approach by implementing an access control type system as an instance of a pluggable verification module.

4.2 Logical Verification of Security Properties

Logical Verification Techniques. Research into program logics has a long history, dating back to the seminal work by Floyd [43] and Hoare [51] on program logics and by Dijkstra [37] on weakest precondition calculi in the late 1960s and early 1970s. There has been steady progress since these early days, resulting in tool-supported program logics for realistic programming languages. In particular the last 10 years have seen a burst of activity around Java program verification which has culminated in the realization of verification environments for Java programs. Many of these environments use as specification language the Java

Modeling Language, which we describe below, and can be used for verifying security properties as well as functional properties of Java programs.

Java Modeling Language. The Java Modeling Language [55] (JML) is a behavioral interface specification language designed for Java. It relies on the design by contract approach [69] to guarantee that a program satisfies its specification during runtime. These specifications are given as annotations of the Java source file. More precisely, they are included as special Java comments, either after the symbols //@ or enclosed between /*@ and @*/. For example, the general schema for the annotation of a method is the following:

```
/*@ behavior
  @   requires <precondition>;
  @   ensures <postcondition if no exception raised>;
  @   signals(E) <postcondition when exception E raised>;
  @   assignable <modified fields and variables>;
  @*/
```

where `requires` specifies the conditions on variables, fields and method parameters at the beginning of the method call so that the conditions after `ensures` hold at the end of the method call and the conditions after `signals(E)` hold if an exception is raised and not caught inside the analyzed method. The underlying model is a an extension of Hoare-Floyd logic: if the precondition holds at the beginning of the method call, then postconditions (with and without exceptions) will hold after the call. The `assignable` clause specifies side-effect affected variables and is used during the weakest precondition calculus for method invocations.

Preconditions and postconditions express first-order logic statements, with a syntax following the Java syntax. Thus, they can easily be written by a programmer. The Java syntax is enriched with special keywords: `\result` and `\old(<expr>)` to denote respectively the return value of a method, and the value of a given expression before the execution of the considered method; and `\forall`, `\exists`, `==>` to denote respectively universal quantification, existential quantification and logical implication.

Besides methods specification, it is also possible to annotate a program with class invariants (predicates on the fields of a class that hold at any time in the class) using the keyword `invariant`, loop invariants (inside the code of a method with loops) using the keyword `loop_invariant`, and assertions (that must hold at the given point of the program) using the keyword `assert`.

Finally, when annotating a program, it might be useful to introduce new variables to keep track of certain aspects or computations. Instead of adding them to the program itself, thus adding new code, it is possible to define variables that will only be used for specification. These variables, called *ghost* variables, are defined in a JML annotation with the keyword `ghost` and assigned to a Java expression with the keyword `Set`.

Styles of Specification. Due to its expressiveness and versatility, the JML specification language supports several styles of specifications; the choice of one style

of specification over the others depends on the purpose of the verification effort. In a nutshell, one can either opt for lightweight specifications in which one introduces enough annotations to reason about some specific safety property, such as the absence of exceptions, or heavyweight specifications where functional behavior is considered. There is of course a great liberty in how "lightweight" or "heavyweight" a specification should be, and different styles can be used in different parts of an application.

In addition, one may opt for defensive specifications, in which methods are annotated with preconditions that prevent exceptions to occur, or offensive specifications, which use appropriate clauses to specify exceptional postconditions.

Verification Techniques and Tools. JML specifications correctness can be verified either during runtime or statically [18]. To be verified during runtime, the source code must have been compiled using `jmlc`, which is a enhanced Java compiler for JML annotated code. This compiler adds to the generated program assertions checking instructions corresponding to the JML specifications of the program: preconditions, postconditions and loop or class invariants. An exception is raised during the execution if a JML condition fails. The JML runtime assertion checker can be used for unit testing [26].

For the static verification of Java programs, several tools are available using (variations of) JML as specification language. These tools adopt different compromises between soundness and automation, and thus it is useful to use them in combination, starting from automatic but unsound tools, and pursuing with sound but interactive tools. Among these tools, ESC/Java2 [28] offers the higher level of automation as it does not require any user interaction and relies on the Simplify automatic prover. It is particularly useful for checking null pointers or array bounds limits; however it is unsound and incomplete. Other static verification tools such as JACK [20], Jive [70], Krakatoa [65] and Loop [11] generate proof obligations that can be discharged using proof assistants or automatic provers. These tools are sound but require user interaction.

Model-checking techniques provide yet another means to verify the correctness of Java programs against their JML annotations. Bogor [85] exploits such techniques to provide automatic verification of concurrent Java programs with JML annotations.

Annotation Assistants. Program verification using logics may require substantial amounts of annotations in programs, and the costs of annotating programs can become prohibitively high as programs increase in size. Automated support for annotating programs is of great benefit to allow program verification to scale to larger programs.

There are several tools and techniques for inferring annotations. One technique consists in using weakest precondition calculi, possibly in variant form, to generate defensive specifications that prevent run-time exceptions. Another technique to infer annotations is abstract interpretation, which can be used for instance to infer constraints on the range of integer fields, or loop invariants [77], class invariants [64] or object invariants [25]. A third technique consists in instrumenting an existing static analysis to generate annotations related to the

property checked by the analysis. These techniques are implemented in some of the aforementioned tools, or in separate tools. In the next paragraphs we illustrate how some annotation assistants can be used to specify security properties of Java applications.

Limitations of JML Technology. Despite having shown its usefulness of a variety of case studies, the JML technology is still under development, and many technical issues remain to be solved. For example, JML is currently not appropriate for reasoning on complex data-structures such as linked-lists or trees because no global property on these structures can be stated in JML. This limitation of JML is related to the first-order logic on which JML is based and that prevents complex quantification over structures or predicates. Also, JML does not allow to in specifications external functions written in a back-end tool, or to use back-end tools to reason about program termination. Another severe restriction of current JML verification tools is the limited support they provide for reasoning about concurrent programs. There are however some ongoing efforts to extend JML with support for reasoning about multi-threaded programs [86].

A final drawback of the JML approach and of the weakest precondition calculus is the difficulty of controlling the shape and size of generated proof obligations. While some techniques have been developed to avoid the size of proof obligations, verifying the functional correctness of some programs, even of modest size, can lead to very complex and unpalatable proof obligations.

High-Level Security Properties

Security Rules. When programming applications using the Java technology, developers are required to follow security rules that pertain to the programming idioms used for developing the applications, and that are intended to complement the security mechanisms provided by the platform. Such security rules cover a wide range of properties, including safety policies not guaranteed by the platform, security properties related to confidentiality or resource control, as well as properties that ensure a correct and legitimate usage of the API. Examples of such properties are detailed below.

Exception Safety. Exception safety is an important concern when developing applications, and guidelines for programming Java applications often consider several forms of exception safety. For example:

- *No runtime exceptions at top-level.* The presence of runtime exceptions at top-level constitutes one common form of programming errors that is undetected by the JavaCard platform security mechanisms. If raised, such exceptions should be caught during the program execution (unlike ISO exceptions whose presence at top-level should not be considered as a programming error).
- *No uncaught exception in transactions.* A first step towards well-formed transactions, defined below, is to guarantee that exceptions do not escape transactions (i.e. that exceptions thrown inside a transaction should be caught inside this transaction).

API Usage. While a careful design of the API is crucial for constraining the interactions between applications and its environment on the card, the security of the device can only be guaranteed if applications perform a correct and legitimate usage of the API. Therefore, a large number of security rules focus on API usage.

- *Well-formed transactions.* In order to maintain the card in a coherent state, the API provides a transaction mechanism that guarantees the atomicity of updates performed within transactions, see Section 2.2. This rule requires that all calls to `beginTransaction` are matched by exactly one call to `commitTransaction` or `abortTransaction`, and conversely.
- *Bounded retries.* No authentication may happen within a transaction. The rule is designed to deter attacks that exploit the atomicity properties of transactions in combination with timing leaks in the implementation of the authentication mechanisms. In a nutshell, if authentication is done within a transaction and response times to authentication challenges are longer in case of a negative reply (say t_0 for a positive reply and $t_0 + t$ for a negative reply), it is possible for the user to pull out the card between t_0 and $t_0 + t$ in the absence of reply from the card (and hence if authentication failed). By pulling out the card, the user forces the card to roll back to the state prior to starting the transaction, and in particular to reset the retry counter. Hence the user allows himself an unbounded number of retries. Beyond its anecdotal nature, this example illustrates that security rules can be crafted to account for vulnerabilities that would be difficult to capture at a more semantical level.

Security guidelines for application developers may also include more specific security rules about API usage, including:

- legitimate and controlled use: instances of such rules include forbidding or restricting calls to given methods, e.g. forbidding GSM applications to call the method that trigger the sending of SMS messages, or requiring that calls to such methods are only performed in authenticated mode;
- privacy: in case of a GSM application requiring access to some positioning system (such as Global Positioning System GPS) to customize their services, security rules may be set to guarantee the application does not divulge the location of the phone and its owner to an unauthorized parties.

While falling short of providing a solid foundation to software security, such rules provide partial guarantees for properties that are difficult to capture formally, and are important in practice because they embody the know-how of security experts.

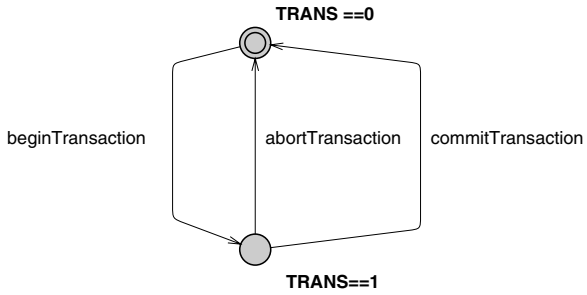
Enforcing Security Rules. Verification techniques based on JML provide an effective means to enforce many security rules for Java applications. In particular, M. Pavlova *et al* [79] propose a general method for guaranteeing a correct usage of the API:

1. the developer express the security rule in some appropriate formalism. Following an established practice popularized by S. Schneider [89], security properties are expressed as automata that constrain the usage of API methods. Intuitively, security automata specify at an abstract level, and independently of any program, preconditions that must hold at certain points during execution[3];
2. JML annotations are synthesized from the automaton and attached to those API methods whose behavior is specified by the automaton. Then JML annotations are propagated throughout the program, and proof obligations are generated from the annotated program. The propagation of the JML annotations is performed by some ad-hoc propagation algorithm, and proof obligations are generated by a verification condition generator. Currently, the propagation of annotations and the generation of proof obligations is performed in two different passes, although it is possible to perform both tasks in a single task due to the similarities between the verification condition generator and the propagation algorithm;
3. proof obligations are dispatched to automatic provers or interactive proof assistants, where they can be discharged by decision procedures or through user interaction. If all proof obligations can be discharged, then the application respects the security property. As a corollary, it follows that inserting runtime checks to monitor that the application only performs "allowed transitions" is superfluous.

The method has been implemented in the JACK verification environment and applied to several case studies from the smartcard domain. The overall conclusion is that the method contributes to carrying out formal security analyses and to improving the quality of applications, while remaining reasonably accessible to developers which are not familiar with formal techniques. Nevertheless the method is by no means complete. For example, B. Jacobs, C. Marché and N. Rauch [52] report on an experiment that goes beyond the technique of [79] and involves a substantial amount of work, both in terms of (non-automated) specification and interactive verification, for checking exception safety of a commercial smartcard applet.

Example. We illustrate how the method can be used to enforce atomicity properties, and in particular that transactions are well-formed. In order to specify that transactions are well-formed, we use an automaton involving one variable `TRANS` initialized to 0 and used to record whether or not there is an ongoing transaction, that corresponds to the two states `TRANS==0` and `TRANS==1`. Then the automaton contains transitions that correspond to correct calls of the methods for beginning, committing and aborting a transaction:

[3] This first step of the method is not reported in *loc cit*, its implementation being posterior to its publication.



The first step towards verifying the well-formedness property is to declare the variables that appear in the automaton with their initial values, here:

```
/*@ static ghost int TRANS == 0; @*/
```

and to generate so-called core annotations for the API methods that appear in the automaton, which in the case of the method `beginTransaction` is:

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 1; @*/
public static native void beginTransaction()
                          throws TransactionException;
```

while in the case of `commitTransaction` the core annotation is:

```
/*@ requires TRANS == 1;
  @ assignable TRANS;
  @ ensures TRANS == 0; @*/
public static native void commitTransaction()
                          throws TransactionException;
```

Then, the second step is to propagate the annotations in the application being verified. We consider two code fragments.

In the first code fragment, we have a method `m`, whose only method calls are those shown, and which does not contain any set annotations.

```
public static void m() { ... // some computations
           JCSystem.beginTransaction();
           ... // some computations within transaction
           JCSystem.commitTransaction(); }
```

The problem is to annotate `m` in such a way that the precondition of `m` guarantees the precondition of `beginTransaction`, and dually that the postcondition of `m` follows from the postcondition of `commitTransaction`. As we assume that `TRANS` is not modified by the code that precedes the call to `beginTransaction` or that follows the call to `commitTransaction`, the precondition of `m` is inherited from that of `beginTransaction`, and the postcondition of `m` is inherited from that of `commitTransaction`. Our algorithm to propagate annotations will generate exactly such a specification:

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 0; @*/
public static void m() { ... // some computations
           JCSystem.beginTransaction();
           ... // some computations within transaction
           JCSystem.commitTransaction(); }
```

Note that one could device finer propagation algorithms that account for the computations performed by `m`, but the resulting specification of `m` would become cluttered. Upon completion of the annotation phase, one must proceed with the verification of the annotated method. For the method above, the proof obligations will include a proof obligation that the postcondition of `m` is guaranteed by the postcondition of `commitTransaction`, a proof obligation that the precondition of `m` ensures the precondition of `beginTransaction`, and a proof obligation that the postcondition of `beginTransaction` ensures the precondition of `commitTransaction`. These proof obligations can be discharged automatically.

The second code fragment aims at illustrating that verification environments can also provide mechanisms to trace the potential source of violations of security properties. If we apply our method to the code fragment:

```
public static void m() { ... // some computations
        JCSystem.beginTransaction();
        ... // some computations within transaction
        JCSystem.beginTransaction();
        ... // some computations within nested transaction
        JCSystem.commitTransaction();
        ... // some computations within transaction
        JCSystem.commitTransaction(); }
```

we shall obtain 5 proof obligations including a proof obligation that (in essence) `TRANS == 1` implies `TRANS == 0`, which corresponds to the proof obligation that ties the postcondition of (the first call to) `beginTransaction` with the precondition of (the second call to) `beginTransaction`. Verification environments such as Jack highlight the code fragment that corresponds to the unprovable proof obligation, and thus indicate the source of the violation of the proof obligation. As pointed in [61], the verification condition generation mechanism of Jack is particularly well suited for linking unprovable proof obligations with problematic code fragments, but the authors of [61] show that it is possible to achieve a similar effect for other forms of verification condition generators that avoid generating an exponential number of proof obligations [42].

Secure Information Flow. Due to its particular nature, a major concern of smartcards applications is to guarantee confidentiality and integrity of data. Section 4.1 advocates to address confidentiality issues from the perspective of non-interference, and indicates that information flow type systems can be used

for enforcing non-interference. However, information flow type systems are extremely conservative and reject many secure programs.

Thus several works such as [32,6] suggest the use of program logics such as dynamic logic or Hoare logic to verify non-interference. In these frameworks, non-interference is reduced to a property about a single program execution using self-composition: basically, a program P is non-interfering if, given two sets of inputs that only differ by the values of secret data, the successive execution of P for each of these set leads to observably equal public outputs.

More formally, as presented in [6], let $\vec{l}$ and $\vec{h}$ be the sets of respectively public (low) and secret (high) variable of a simple (without mutable structures) imperative program P, let $\vec{l'}$ (resp. $\vec{h'}$) be a renaming with fresh names of the variables of $\vec{l}$ (resp. $\vec{h}$), then $P(\vec{l},\vec{h})$ is non-interferent if before the execution $\vec{l}$ and $\vec{l'}$ are such that $\vec{l}=\vec{l'}$ then after the execution of $P(\vec{l},\vec{h});P(\vec{l'},\vec{h'})$, where ; is the usual sequential composition, the equality $\vec{l}=\vec{l'}$ still holds. Or equivalently, the following Hoare triple must be valid:

$$\{\vec{l}=\vec{l'}\}P(\vec{l},\vec{h});P(\vec{l'},\vec{h'})\{\vec{l}=\vec{l'}\}$$

Although the above definition of non-interference with self-composition uses Hoare logic, verifying this property for a given program is not possible with standard JML tools. Indeed, specifications relate to the self-composed program, not the original one, make use of renamed variables and special care must be taken for method invocation. In [39], the Krakatoa tool and JML have been extended to allow specifications for non-interference, without changing the underlying weakest precondition calculus of the tool.

As mentioned in Section 4.1, non-interference is too strict since it forbids any computation on secret variables observable on public variables, even those expected for the normal behavior of smartcards, such as PIN code verification. One advantage of JML is that it provides a precise relationship between inputs and outputs of the program. Thus, it becomes possible to capture information release using JML. An example of information release, expressed with the extended version of JML presented in [39] is given below. This example deals with PIN code verification. The secret variable `pin` stores the actual PIN code, the public input variable `in` represents the attempted code, and the public output variable `acc` reveals whether `in` and `pin` agree. The specification of non-interference with declassification follows: tested with two different sets of input that are both valid or both invalid tries, the algorithm must give identical result for the access.

```
/*@ public normal_behavior
  @ requires_ni (\ni1(in)==\ni1(pin))<==>(\ni2(in)==\ni2(pin))
  @ ensures_ni \ni1(acc) == \ni2(acc);
  @*/
void pin_verification(int in) {
  if (in == pin)
    acc = true;
  else
    acc = false;
}
```

In this code, `\ni1(<var>)` (resp. `\ni2(<var>)`) correspond to the renamed variables that appear in self composition, `<==>` is logical implication, and `requires_ni` (resp. `ensures_ni`) is a keyword to introduce pre-condition (resp. post-condition) of the Hoare triple in self composition specifications.

Resource Consumption. Controlling resource consumption of downloaded applications is a compulsory measure for preventing denial of service attacks on devices with restricted resources. Yet current security architectures do not provide any mechanism to control resource consumption. In this section, we illustrate how JML can be used to specify and verify memory usage of Java applications.

Principles. Verification techniques based on JML are also appropriate for performing a precise analysis of resource consumption for Java programs. The basic idea, described in [8], consists in:

1. using a ghost variable `Mem` that provides an upper bound for the memory consumed by the program at any given program point;
2. attaching to each method `m` a postcondition that predicts its memory consumption. The prediction can be express at different levels of granularity. For example, the estimate may be global, or it may account for the value of inputs, or it may distinguish between normal and abnormal termination, or between the different types of memory;
3. verifying for each method that the predicted memory consumption is indeed an upper bound of the memory consumption, by inserting immediately after every bytecode that allocates memory a ghost assignment that increments of `Mem` by the amount of memory consumed by the allocation, and verifying the resulting annotated program.

The correctness of the approach can be derived from the correctness of logical verification techniques. As a consequence, every run of a method will not consume more than `km` memory units, provided the corresponding annotated method has been proved correct. Of course, the correctness of the approach hinges on the fact that the programmer correctly specified the amount of memory consumed by each allocation.

The approach is practical in that it allows to specify and enforce precise memory consumption policies for applications that may involve such features as exceptions, subroutines, and recursive methods. Its practicality can be further enhanced by annotation assistants which infer appropriate JML annotations about memory consumption. Of course, such assistants are necessarily incomplete an user interaction may be required for exploiting the full power of JML. In addition, user interaction can be required to discharge proof obligations generated from the annotated applet.

While our approach is purely static, L.-A. Fredlund [46] suggests to use a runtime monitor to control the execution of JavaCard applets whereas A. Chander *et al* [24] propose to control memory consumption with an hybrid approach that combines static and dynamic verifications. A further difference between

our approach and that of Chander *et al* is that they do not require user interaction; while avoiding user interaction does in principle restrict the scope of their method, they show the applicability and effectiveness of their tool on a substantial example.

Example. One can specify and verify that the method `m` below will not use more than `km` memory units

```
public void m(A a) {
  if (a == null) {
    a = new A();
  }
  a.b = new B();
}
```

by annotating the program as follows

```
//@ ensures Mem <= \old(Mem) + km;
public void m(A a ) {
  if (a == null) {
    a = new A();
    //@ set Mem += ka;
  }
  a.b = new B();
  //@ set Mem += kb;
}
```

and verifying that the annotated method is correct provided `ka`+`kb` $\leq$ `km`. Due to the simplicity of the code (e.g. it does not contain loop nor recursive methods), the annotation assistant is able to infer the specification of the method, once given (over)estimates `ka` and `kb` of the memory consumed by the allocation of an instance of class `A` and `B` respectively.

Other Quantitative Properties. Due to their constrained resources, trusted personal devices are very vulnerable to denial-of-service attacks, and it is therefore important to control many facets of the resource usage made by applications. In addition to memory consumption, which is discussed above, security policies may concern such resources as library calls, communication channels, bandwidth, and power consumption. For some resources, it is possible to adapt the technique described above.

Verification from the Code Consumer Perspective. Throughout this section, the problem of applet validation has been addressed from the perspective of the application developer who wants to gain confidence in the quality of the software it develops. Thus, we have focused on applet validation at source code level. Such a focus has been quite common in the area of smartcards, where currently the card issuer has control over the applications loaded on the card, and is likely to have access to the applications source code.

The situation is somewhat different in other application areas of trusted personal devices, and in particular in the area of telecommunications where

operators are faced with the possibility of offering their customers new services by deploying applications originating from an untrusted software company. The issue here is that operators, while worried about the negative impact on business if the code is malicious or simply erroneous, do not have access to the source code of applications both for intellectual property reasons (the software company does not want to disclose its source code). In such a scenario, the issue arises of verifying bytecode applications.

Fortunately, logical techniques such as those presented above are also applicable to bytecode level, and can be used by the code consumer to check that the application that he downloads is secure. For example, C. Quigley [84] considers program logics for Java bytecode, as do F. Bannwart and P. Müller [3]. Further, M. Wildemoser and T. Nipkow [95] have formalized in Isabelle/HOL a verification condition generator for a fragment of Java bytecode, and shown its correctness w.r.t. an operational semantics. Motivated by the prospect of bringing the benefits of source code verification to the code consumers, L. Burdy and M. Pavlova [19] have also developed a verification condition generator for Java bytecode programs. The verification condition generator, which has been integrated in JACK, outputs proof obligations from extended class files that can either be obtained manually by inserting annotations into user-defined attributes, or by a compiler that compiles JML annotated Java programs into extended class. In both cases, annotations are written in the Bytecode Modeling Language (BML), which may be considered as the counterpart of JML for bytecode. Rather similar work for C# has taken place in the context of the Spec# project [4], which has defined an extension of C# with annotations and type support for nullity discrimination. Such annotated programs are then compiled with their specifications to extended .NET files, which can be run using the .NET platform. Specifications are checked at run-time or verified using the Boogie static checker.

Bringing the benefits of source code verification to code consumers not only requires to develop logical methods to reason about bytecode programs, but also to develop mechanisms that allow code consumers to check efficiently that programs are correct. In their work of Proof Carrying Code, G. Necula and P. Lee [75,74] propose a general solution to this problem by requiring that components should come equipped with a certificate which can be used by the consumer to verify statically that the components are correct. They also advocate the idea of certifying compilation [76], where the focus is on safety properties which can be proven automatically through an extended compiler that synthesizes annotations from the information it gathers about a program, and a checker that discharges proof obligations generated by the verification condition generator.

Some of the security properties described above cannot be verified automatically without a loss of precision, and thus as a result of favoring automatic verification, certifying compilation does not fully exploit the flexibility and expressiveness provided by logical verification techniques. One complementary approach to certifying compilation consists in exploiting the results of source code

verification, and constructing certificates for bytecode programs from certificates for the corresponding source code programs. In some recent unpublished work, G. Barthe, T. Rezk and co-workers have showed for a restricted fragment of Java that proof obligations are almost preserved by non-optimizing compilers; thus in this context it is possible to reuse certificates almost directly. Preservation of proof obligations by compilation is destroyed by simple program optimizations; nevertheless preliminary experiments suggest that it is possible to construct from certificates for source code programs certificates for bytecode programs obtained by optimizing compilation.

Types vs. Logics. While Section 4.1 focused on the use of type systems to guarantee that applications respect security policies related to confidentiality and memory consumption, this section shows that logical verification methods can be used to enforce the same policies. The main benefits of using logical verification techniques are: expressiveness (logics support customizable security policies), versatility (the same logic can be used for a great variety of analyses, thus allowing analyses to be combined), and precision (logics can be used to provide precise statements of program behavior).

In order to improve the quality of their static analyses, several researchers are now exploring the possibility of making a systematic use of logical verification methods to verify program properties that are traditionally checked by static analyses, for example ownership [35]. This line of work seems promising, and opens the perspective for precise static analyses. However such program analyses based on logic may require user interaction (for discharging the proof obligations generated by the analysis of the program), especially if they seek to exploit the full power of logic in an analysis that combines expressiveness and precision. Unfortunately, requiring substantial user interaction is an obstacle to the scalability of this method.

On the other hand, type systems benefit from a combination of three important features: simplicity (types can be viewed as a particularly simple form of assertions), automation (type systems compute a decidable approximation of the property to be enforced) and scalability (type systems are compositional and allow to reduce the verification of a complex program into simpler verification tasks). They are thus amenable to on-device checking. In order to get the best of both worlds, it seems therefore relevant to gain a more systematic understanding of the relationship between program logics and type systems, and eventually to combine both verification methods in a single technique that remains precise while avoiding excessive user interaction.

Application Correctness. Enforcing security properties of JVM applications is an important step towards guaranteeing the security of trusted personal devices. In most situations, verifying that applications adhere to a security policy will provide sufficient guarantees (under the assumption that the platform is correctly implemented). In some situations, one may however be interested in achieving a higher degree of reliability by showing that an application, or a fragment of it, has the expected functionality. As a fully fledged behavioral interface

specification language, JML offers the possibility to verify and specify the functional behavior of Java applications, and many of the JML tools used for applet validation can also be used to verify functional specifications.

The most immediate application of functional verification is to establish that some fragment of a Java application is correct w.r.t. its intended behavior. For example, C.B. Breunesse *et al* [17] show that decimal arithmetic is correctly implemented in an electronic purse application. Below we briefly describe two further applications, and point to some of the issues involved.

JavaCard API. Section 3.3 provides a general methodology to reason about the Java API, for example to prove that some method from the API has some expected functional behavior. Such a verification is valuable from the point of view of guaranteeing the correct design of the API, but leaves open the issue of the implementation of the API, which may not coincide with its description in the formal model. Therefore one must ensure that the API is correctly implemented. Since a substantial part of the API is implemented in Java, it is possible to use JML and its associated verification techniques for this purpose.

E. Poll and its co-workers [12,68] have developed reference JML specifications of the JavaCard API, and verified formally the correctness of some API methods against their specification. For example, they establish the correctness of the methods in the class AID, and in particular of the method `getBytes` which is called to get the AID bytes encapsulated within AID object. The method takes two parameters `dest` which represents the byte array to copy the AID bytes, and `offset` which represents within `dest` where the AID bytes begin, and returns the length of the AID bytes. Its code and specification are given in Figure 5. The JML annotations aims at specifying that bytes representing the AID are accurately copied to the right position in the given array `dest` and at constraining the behavior of the method if an exception is raised (by `arrayCopy`). The corresponding proof obligations will directly follow from the specifications of `arrayCompare` and `arrayCopy` used in the annotations and in the code.

The implementation of the method `getBytes` contains a call to the native method `arrayCopy`; in order to reason about the correctness of the method `getBytes`, one must therefore dispose of a JML specification of `arrayCopy`. As native methods are not implemented in Java, we cannot use JML tools to verify them against their specifications, and thus we must trust that the specification of native methods is faithful to the implementation. Thus the correctness of the method `getBytes` will be established under the assumption that the specification of the native method is correct. The correctness proof is easy; it should be pointed however that generating the verification conditions that guarantee the correctness of the method involves subtle semantical issues, including the semantics of method calls in specifications [33].

Java-Based Components. Trusted personal devices are evolving from dedicated devices with a specific usage into general purpose devices that must provide users with a uniform access to multiple services. Such an evolution raises some

```
/*@ public behavior
  @ requires dest != null;
  @ requires dest != theAID;
  @ requires 0 <= offset;
  @ requires offset+theAID.length <= dest.length ;
  @
  @ assignable dest[offset..offset+theAID.length-1];
  @
  @ ensures \result == theAID.length;
  @ ensures Util.arrayCompare(theAID,(short)0,
  @         dest,offset,(short)theAID.length) == 0;
  @
  @ ensures
  @         (\forall short i; 0 <= i && i < theAID.length
  @                         ==> theAID[i] == dest[offset+i]);
  @
  @ signals (NullPointerException) dest == null;
  @ signals (ArrayIndexOutOfBoundsException) dest != null;
  @ signals (ArrayIndexOutOfBoundsException)
  @         (0>offset || offset + theAID.length > dest.length);
  @*/

public byte getBytes (byte[] dest, short offset)
  throws ArrayIndexOutOfBoundsException, NullPointerException {
  Util.arrayCopy(theAID, (short) 0,
                 dest, offset, (short) theAID.length);
  return (byte) theAID.length;
}
```

Fig. 5. JML specification of getBytes method from AID class

technical difficulties: in particular, trusted personal devices are heterogeneous in their computational infrastructure (operating systems, communication protocols, libraries) and resources (memory, power autonomy, connectivity), and due to physical, technological and economical constraints, it is not possible to install on such devices all functionalities that may be needed *a priori* by potential applications. One promising solution to overcome this limiting factor is to embed on devices extensible virtual machines that can be extended with the computational infrastructure, platform or libraries, needed to execute the required services. Such a solution also enables remote system upgrades that allow device issuers to perform maintenance over the network in situations that would otherwise require devices to be recalled. Such remote updates are an attractive possibility (although there are some concerns about negative implications of issuers having remote control over devices) and form an integral part of the deployment model in application domains such as telephone networks and digital video infrastructures. The adoption of a component-based approach in the development of runtime environments and security architectures for trusted personal devices, and the perspective of downloading such components dynamically, raise important security issues as the security of a device might be fatally com-

promised upon installing a faulty or malicious component. In such instances, it becomes important to be assured of the functional correctness of incoming programs prior to their loading on device.

Many current projects to develop modular virtual machines adopt Java as their programming language [94,54,78], making it conceivable to use JML verification tools to establish the functional correctness of key components. Preliminary experiments suggest that it is feasible to prove component correctness in toy examples, e.g. to prove the correctness of a bytecode verifier for a toy assembly language. However, even dealing with toy examples involves a number of difficulties that range from JML semantics to scalability of JML verification techniques. Under such circumstances, proving formally the correctness of a realistic component remains beyond current state of the art, and an exciting challenge for JML verification technology.

5 Perspective

Smart cards have proved an ideal application domain for formal methods, and the last few years have seen substantial achievements both in the area of platform verification and application validation. In spite of such scientific progress, formal methods have not gained a widespread acceptance in industry, and the use of formal methods in the smart cards and trusted personal devices industry is quite often confined to R&D laboratories. According to a recent roadmap [1], cost effectiveness and scalability remain two bottlenecks for a wider use of formal methods in the smartcard industry; see also [59] for another industrial perspective on this issue. Addressing these bottlenecks is an important engineering challenge to be tackled by the formal methods community.

In addition, the increasing complexity and connectivity of trusted personal devices raise new challenges and opportunities for formal methods. Some specific technical challenges have already been mentioned in this chapter: extending platform verification and applet verification techniques to cover multi-threading, developing a systematic encoding of security properties in interface specification languages such as JML, integrating type systems and logical verification techniques, establishing the functional correctness of system components. These specific challenges will serve as stepping stones towards the much greater challenge of achieving a better integration of formal methods in security architectures for trusted personal devices, and eventually security architectures for large networks of Java-enabled devices. The forthcoming European project "MOBIUS: Mobility, Ubiquity, and Security" [83] will aim at addressing many of these challenges and opportunities in the context of large and distributed networks of Java-enabled devices that aim at providing services globally, uniformly, and securely.

Acknowledgments. The authors are partially supported by the European IST project INSPIRED, the RNTL project CASTLES and the ACI Sécurité SPOPS and GECCOO. The work reported here has been partially funded by the Euro-

pean IST projects PROFUNDIS and VERIFICARD, by the European thematic network RESET, and by the INRIA research actions S-Java and MODOCOP.

The authors are grateful to Julien Forest and Tamara Rezk for their comments on an earlier version of the chapter, and to the former and present members of the EVEREST and LEMME teams at INRIA Sophia-Antipolis for stimulating interactions over the years.

References

1. Roadmap for European Research on Smartcard Technologies. `http://www.ercim.org/reset`
2. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.
3. F. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Proceedings of Bytecode'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005.
4. M. Barnett, K.R.M. Leino, and W. Schulte. The spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 50–71. Springer-Verlag, 2005.
5. G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Jakarta: tool-assisted specification and verification of the JavaCard Platform. *Journal of Automated Reasoning*, 2006. To appear.
6. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.
7. G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. In *Proceedings of FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 99–113. Springer-Verlag, 2004.
8. G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In B. Aichernig and B. Beckert, editors, *Proceedings of SEFM'05*. IEEE Press, 2005.
9. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
10. D. Basin, S. Friedrich, and M. Gawkowski. Bytecode Verification by Model Checking. *Journal of Automated Reasoning*, 30(3-4):399–444, December 2003.
11. J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.
12. J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'00*, volume 2041 of *Lecture Notes in Computer Science*, pages 137–150. Springer Verlag, 2001.
13. F. Besson, T. Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proceedings of PPDP'02*, pages 76–87. ACM Press, 2002.
14. F. Besson, T. Jensen, D. Le Métayer, and T.Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

15. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
16. B. Blanchet. Escape analysis for java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.
17. C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and Verifying a Decimal Representation in Java for Smart Cards. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 304–318. Springer-Verlag, 2002.
18. L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005. To appear.
19. L. Burdy and M. Pavlova. Annotation carrying code. Manuscript, 2005.
20. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME'03*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
21. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Proceedings of FM'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
22. D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java card object sharing. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2001.
23. L. Casset, L. Burdy, and A. Requet. Formal Development of an Embedded Verifier for JavaCard ByteCode. In *Proceedings of DSN'02*. IEEE Computer Society, 2002.
24. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag, 2005.
25. B.-Y.E. Chang and K.R.M. Leino. Inferring object invariants. In A. Cortesi and F. Logozzo, editors, *Proceedings of AIOOL'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005. To appear.
26. Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP'02*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, 2002.
27. A. Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, 2004.
28. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML — progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
29. Connected Limited Device Configuration (CLDC) and the K Virtual Machine (KVM). `http://java.sun.com/products/cldc`
30. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
31. Common Criteria. `http://www.commoncriteria.org`
32. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Informal proceedings of WITS'03*, 2003.

33. A. Darvas and P. Müller. Reasoning About Method Calls in JML Specifications. Manuscript, 2005.
34. D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *Proceedings of WIESS'02*. Usenix Association, 2002.
35. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.
36. W. Dietl, P. Müller, and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 129–150. Springer-Verlag, 2005.
37. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
38. G. Dufay. *Vérification formelle de la plateforme JavaCard*. PhD thesis, Université de Nice Sophia-Antipolis, 2003.
39. G. Dufay, A. Felty, and S. Matwin. Privacy-Sensitive Information Flow with JML. In R. Nieuwenhuis, editor, *Proceedings of CADE'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. To appear.
40. M. Eluard and T. Jensen. Secure object flow analysis for java card. In *Proceedings of CARDIS'02*, pages 97–110. USENIX Association, 2002.
41. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI'03*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.
42. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of POPL'01*, pages 193–205. ACM Press, 2001.
43. R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
44. P. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Proceedings of OOPSLA'04*, pages 404–418. ACM Press, 2004.
45. P. Fong and R. Cameron. Proof linking: modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 9(4):379–409, October 2000.
46. L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In K. Havelund and G. Rosu, editors, *Proceedings of RV'04*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 217–233. Elsevier Publishing, 2004.
47. 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.
48. 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.
49. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
50. T. Higuchi and A. Ohori. A static type system for JVM access control. In *Proceedings of ICFP'03*, pages 227–237. ACM Press, 2003.
51. C. A. R. Hoare. An axiomatic basis for computer programming. *Commununications of ACM*, 12(10):576–580, 1969.
52. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of AMAST'04*, volume 3116 of *Lecture Notes in Computer Science*, pages 241–257. Springer-Verlag, 2004.
53. JavaCard Technology. `http://java.sun.com/products/javacard`

54. Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net/
55. JML Specification Language. http://www.jmlspecs.org.
56. J.W. Jo, B.M. Chang, K. Yi, and K.M. Choe. An uncaught exception analysis for Java. *Journal of systems and software*, 72(1):59–69, 2004.
57. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.
58. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
59. J.-L. Lanet. Are smart cards the ideal domain for applying formal methods? In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proceedings of ZB'2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 363–374, 2000.
60. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290(1):741–778, October 2002.
61. K.R.M. Leino, T. Millstein, and J.B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55:209–226, March 2005.
62. X. Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.
63. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
64. F. Logozzo. Automatic inference of class invariants. In G. Levi and B. Steffen, editors, *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer-Verlag, 2004.
65. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard Programs annotated with JML Annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
66. R. Marlet and D. Le Métayer. Security properties and java card specificities to be studied in the secsafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
67. C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Proceedings of MMMACNS*, volume 2052 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
68. H. Meijer and E. Poll. Towards a full formal specification of the java card. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 165–178. Springer-Verlag, 2001.
69. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.
70. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from sct.inf.ethz.ch/publications, 2000.
71. M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
72. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
73. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
74. G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
75. G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

76. G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344, 1998.
77. J.W. Nimmer and M.D. Ernst. Automatic generation of program specifications. In *Proceedings of ISSTA'02*, volume 27, 4 of *Software Engineering Notes*, pages 232–242. ACM Press, 2002.
78. OVM project. `http://www.ovmj.org/`
79. M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*. Kluwer, 2004.
80. Global Platform. See `http://www.globalplatform.org`
81. J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Proceedings of ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.
82. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, March 2005.
83. Mobius Project. `http://mobius.inria.fr`
84. C.L. Quigley. A Programming Logic for Java Bytecode Programs. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.
85. Robby, E. Rodríguez, M.B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model-checking framework. In K. Jensen and A. Podelski, editors, *Proceedings of TACAS'04*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420, 2004.
86. E. Rodriguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending jml for modular specification and verification of multi-threaded programs. In *Proceedings of ECOOP'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
87. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
88. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press, 2005.
89. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.
90. G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, 2004.
91. I. A. Siveroni. Operational semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, 2004.
92. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
93. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.
94. Java In the Small Project. `http://www.lifl.fr/rd2p/jits/`
95. M. Wildmoser and T. Nipkow. Asserting bytecode safety. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2005.
96. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.
97. H. Xi and S. Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proceedings of CASCOON'99*, pages 110–125, November 1999.