Bytecode verification on Java smart cards

Xavier Leroy

Bytecode verification on Java smart cards

Xavier Leroy *†

Draft of an article to appear in Software Practice & Experience, 2002

Abstract

This article presents a novel approach to the problem of bytecode verification for Java Card applets. By relying on prior off-card bytecode transformations, we simplify the bytecode verifier and reduce its memory requirements to the point where it can be embedded on a smart card, thus increasing significantly the security of post-issuance downloading of applets on Java Cards. This article describes the on-card verification algorithm and the off-card code transformations, and evaluates experimentally their impact on applet code size.

Keywords: Bytecode verification; Java; Java Card; smart cards; applets; security.

1 INTRODUCTION

Smart cards are small, inexpensive embedded computers that are highly secure against physical attacks. As such, they are ubiquitous as security tokens in a variety of applications: credit cards, GSM mobile phones, medical file management, ...

Traditionally, smart cards run only one proprietary application, developed in C or assembler specifically for the smart card hardware it runs on, and impossible to modify after the card has been issued. This closed-world approach is being challenged by new, open architectures for smart cards, such as Multos and Java Card.

The Java Card architecture [5] bring three major innovations to the smart card world: first, applications are written in Java and are portable across all Java cards; second, Java cards can run multiple applications, which can communicate through shared objects; third, new applications, called *applets*, can be downloaded on the card post issuance.

These new features bring considerable flexibility to the card, but also raise major security issues. A malicious applet, once downloaded on the card, can mount a variety of attacks, such as leaking confidential information outside (e.g. PINs and secret cryptographic keys), modifying sensitive information (e.g. the balance of an electronic purse), or interfering with other honest applications already on the card, causing them to malfunction.

The security issues raised by applet downloading are well known in the area of Web applets, and more generally mobile code for distributed systems [30, 15]. The solution put forward by the Java programming environment is to execute the applets in a so-called "sandbox", which is an insulation layer preventing direct access to the hardware resources and implementing a suitable access control policy [8]. The security of the sandbox model relies on the following three components:

*Trusted Logic, 5, rue du Bailliage, 78000 Versailles, France. E-mail: Xavier.Leroy@trusted-logic.fr

†INRIA Rocquencourt, domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France. E-mail: Xavier.Leroy@inria.fr

1. Applets are not compiled down to machine executable code, but rather to bytecode for a virtual machine. The virtual machine manipulates higher-level, more secure abstractions of data than the hardware processor, such as object references instead of memory addresses.

2. Applets are not given direct access to hardware resources such as the serial port, but only to a carefully designed set of API classes and methods that perform suitable access control before performing interactions with the outside world on behalf of the applet.

3. Upon downloading, the bytecode of the applet is subject to a static analysis called bytecode verification, whose purpose is to make sure that the code of the applet is well typed and does not attempt to bypass protections 1 and 2 above by performing ill-typed operations at run-time, such as forging object references from integers, illegal casting of an object reference from one class to another, calling directly private methods of the API, jumping in the middle of an API method, or jumping to data as if it were code [9, 31, 14].

The Java Card architecture features components 1 and 2 of the sandbox model: applets are executed by the Java Card virtual machine [29], and the Java Card runtime environment [28] provides the required access control, in particular through its "firewall". However, component 3 (the bytecode verifier) is missing: as we shall see later, bytecode verification as it is done for Web applets is a complex and expensive process, requiring large amounts of working memory, and therefore believed to be impossible to implement on a smart card.

Several approaches have been considered to palliate the lack of on-card bytecode verification. The first is to rely on off-card tools (such as trusted compilers and converters, or off-card bytecode verifiers) to produce well-typed bytecode for applets. A cryptographic signature then attests the well-typedness of the applet, and on-card downloading is restricted to signed applets. The drawback of this approach is to extend the trusted computing base to include off-card components. The cryptographic signature also raises delicate practical issues (how to deploy the signature keys?) and legal issues (who takes liability for a buggy applet produced by faulty off-card tools?).

The second workaround is to perform type checks dynamically, during the applet execution. This is called the defensive virtual machine approach. Here, the virtual machine not only computes the results of bytecode instructions, but also keeps track of the types of all data it manipulates, and performs additional safety checks at each instruction: are the arguments of the correct types? does the stack overflow or underflow? are class member accesses allowed? etc. The drawbacks of this approach is that dynamic type checks are expensive, both in terms of execution speed and memory requirements (storing the extra typing information takes significant space). Dedicated hardware can make some of these checks faster, but does not reduce the memory requirements.

Our approach is to challenge the popular belief that on-card bytecode verification is infeasible. In this article, we describe a novel bytecode verification algorithm for Java Card applets that is simple enough and has low enough memory requirements to be implemented on a smart card. A distinguishing feature of this algorithm is to rely on off-card bytecode transformations whose purpose is to facilitate on-card verification. This algorithm is at the heart of the Trusted Logic on-card CAP file verifier. This product – the first and currently only one of its kind – allows secure execution with no run-time speed penalty of non-signed applets on Java cards.

The remainder of this article is organized as follows. Section 2 reviews the traditional bytecode verification algorithm, and analyzes why it is not suitable to on-card implementation. Section 3 presents our bytecode verification algorithm and how it addresses the issues with the traditional

algorithm. Section 4 describes the off-card code transformations that transform any correct applet into an equivalent applet that passes on-card verification. Section 5 gives preliminary performance results. Related work is discussed in section 6, followed by concluding remarks in section 7.

2 TRADITIONAL BYTECODE VERIFICATION

In this section, we review the traditional bytecode verification algorithm developed at Sun by Gosling and Yellin [9, 31, 14].

Bytecode verification is performed on the code of each non-abstract method in each class of the applet. It consists in an abstract execution of the code of the method, performed at the level of types instead of values as in normal execution. The verifier maintains a stack of types and an array associating types to registers (local variables). These stack and array of registers parallel the operand stack and the registers composing a stack frame of the virtual machine, except that they contain types instead of values.

2.1 Straight-line code

Assume first that the code of the method is straight line (no branches, no exception handling). The verifier considers every instruction of the method code in turn. For each instruction, it checks that the stack before the execution of the instruction contains enough entries, and that these entries are of the expected types for the instruction. It then simulates the effect of the instruction on the operand stack and registers, popping the arguments, pushing back the types of the results, and (in case of "store" instructions) updating the types of the registers to reflect that of the stored values. Any type mismatch on instruction arguments, or operand stack underflow or overflow, causes verification to fail and the applet to be rejected. Finally, verification proceeds with the next instruction, until the end of the method is reached.

The stack type and register types are initialized to reflect the state of the operand stack and registers on entrance to the method: the stack is empty; registers $0, \ldots, n-1$ holding method parameters and the `this` argument if any are given the corresponding types, as given by the descriptor of the method; registers $n, \ldots, m-1$ corresponding to uninitialized registers are given the special type $\top$ corresponding to an undefined value.

Method invocations are treated like single instructions: the number and expected types of the arguments are determined from the descriptor of the invoked method, as well as the type of the result, if any. This amounts to type-checking the current method assuming that all methods it invokes are type-correct. If this property holds for all methods of the applet, a simple coinductive argument shows that the applet as a whole is type-correct.

2.2 Dealing with branches

Branch instructions and exception handlers introduce forks (execution can continue down several paths) and joins (several such paths join on an instruction) in the flow of control. To deal with forks, the verifier cannot in general determine the path that will be followed at run-time. (Think of a conditional branch: at verification time, the argument is known to be of type `boolean`, but it is not known whether it is `false` or `true`). Hence, it must propagate the inferred stack and register types to all possible successors of the forking instruction. Joins are even harder: an instruction

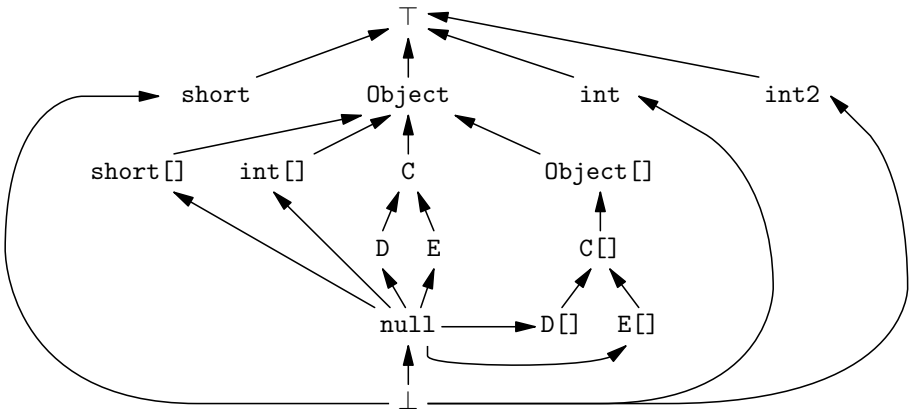

Figure 1: The lattice of types used by the verifier. `C`, `D`, `E` are user-defined classes, with `D` and `E` extending `C`. Not all types are shown.

that is the target of one or several branches or exception handlers can be reached along several paths, and the verifier has to make sure that the types of the stack and the registers along all these paths agree (same stack height, compatible types for the stack entries and the registers).

Sun's verification algorithm deals with these issues in the manner customary for data flow analyses. It maintains a data structure, called a "dictionary", associating a stack and register type to each program point that is the target of a branch or exception handler. When analyzing a branch instruction, or an instruction covered by an exception handler, it updates the type associated with the target of the branch in the dictionary, replacing it by the least upper bound of the type previously found in the dictionary and the type inferred for the instruction. (The least upper bound of two types is the smallest type that is assignment-compatible with the two types. It is determined with respect to the lattice of types depicted in Figure 1.) If this causes the dictionary entry to change, the corresponding instructions and their successors must be re-analyzed until a fixpoint is reached, that is, all instructions have been analyzed at least once without changing the dictionary entries. See [14, section 4.9] for a more detailed description.

The dictionary entry for a branch target is also updated as described above when the verifier analyzes an instruction that "falls through" a subsequent instruction that is a branch target. This way, the dictionary entry for an instruction that is a branch target always contains the least upper bound of the stack and register types inferred on all branches of control that lead to this instruction. Type-checking an instruction that is a branch target uses the associated dictionary entry as the stack and register type "before" the instruction.

Several errors are detected when updating the dictionary entry for a branch target. First, the stack heights may differ: this means that an instruction can be reached through several paths with inconsistent operand stacks, and it causes verification to fail immediately. Second, the types for a particular stack entry or register may be incompatible. For instance, a register contains a `short` on one branch and an object reference on another. In this case, its type is set to $\top$ in the dictionary. If the corresponding value is used further on, this will cause a type error.

Dictionary entries can change during verification, when new branches are examined. Hence, the corresponding instructions and their successors must be re-analyzed until a fixpoint is reached,

that is, all instructions have been analyzed at least once without causing the dictionary entries to change. This can be done efficiently using the standard dataflow algorithm of Kildall [16, section 8.4].

2.3 Performance analysis

The verification of straight-line pieces of code is very efficient, both in time and space. Each instruction is analyzed exactly once, and the analysis is fast (approximately as fast as executing the instruction in the virtual machine). Concerning space, only one stack type and one set of register types need to be stored at any time, and is modified in place during the analysis. Assuming each type is represented by 3 bytes[1], this leads to memory requirements of $3S+3N$ bytes, where S is the maximal stack size and N the number of registers for the method. In practice, 100 bytes of RAM suffice. Notice that a similar amount of space is needed to execute an invocation of the method; thus, if the card has enough RAM space to execute the method, it also has enough space to verify it.

Verification in the presence of branches is much more costly. Instructions may need to be analyzed several times in order to reach the fixpoint. Experience shows that few instructions are analyzed more than twice, and many are still analyzed only once, so this is not too bad. The real issue is the memory space required to store the dictionary. If B is the number of distinct branch targets and exception handlers in the method, the dictionary occupies $(3S+3N+3)\times B$ bytes (the three bytes of overhead per dictionary entry correspond to the PC of the branch target and the stack height at this point). A moderately complex method can have $S=5$, $N=15$ and $B=50$, for instance, leading to a dictionary of size 3450 bytes. This is too large to fit comfortably in RAM on current generation Java cards: a typical 2001 Java card provides 1–2 kilobytes of RAM, 16–32 kilobytes of EEPROM and 32–64 kilobytes of ROM.

Moreover, the number of branch targets B in a method is generally proportional to the size of the method. This means that the size of the dictionary increases linearly with the size of the method, or even super-linearly since the number of registers N is generally increasing too. Consequently, space-saving programming techniques such as merging several methods into a larger one, well established in the Java Card world, quickly result in non-verifiable code even on future smart cards.

Storing the dictionary in persistent rewritable memory (EEPROM or Flash) is not an option, because verification performs many writes to the dictionary when updating the types it contains (typically, several hundreds, even thousands of writes for some methods), and these writes to persistent memory take time (1–10 ms each); this would make on-card verification too slow. Moreover, problems may arise due to the limited number of write cycles permitted on persistent memory.

3 OUR VERIFICATION ALGORITHM

3.1 Intuitions

The novel bytecode verification algorithm that we describe in this article follows from a careful analysis of the shortcomings of Sun's algorithm, namely that a copy of the stack type and register

[1]This figure corresponds to the natural representation for Java Card types: one byte of tag indicating the kind of the type (base type, class instance, array) and two bytes of payload containing for instance a class reference.

type is stored in the dictionary for each branch target. Experience shows that dictionary entries are quite often highly redundant. In particular, it is very often the case that stack types stored in dictionary entries are empty, and that the type of a given register is the same in all or most dictionary entries.

These observations are easy to correlate with the way current Java compilers work. Concerning the stack, all existing compilers use the operand stack only for evaluating expressions, but never store the values of Java local variables on the stack. Consequently, the operand stack is empty at the beginning and the end of every statement. Since most branching constructs in the Java language work at the level of statements (`if...then...else...`, `switch` constructs, `while` and `do` loops, `break` and `continue` statements, exception handling), the branches generated when compiling these constructs naturally occur in the context of an empty operand stack. The only exception is the conditional expression e_1 `?` e_2 `:` e_3, which is generally compiled down to the following JCVM code:

```
        code to evaluate e1
        ifeq lbl1
        code to evaluate e2
        goto lbl2
lbl1:   code to evaluate e3
lbl2:   ...
```

Here, the branch to `lbl2` occurs with a non-empty operand stack.

As regards to registers, many compilers simply allocate a distinct JCVM register for each local variable in the Java source. At the level of the Java source, a local variable has only one type throughout the method: the type τ with which it is declared. In the JCVM bytecode, this translates quite often to a register whose type is initially $\top$ (uninitialized), then acquires the type τ at the first store in this register, and keeps this type throughout the remainder of the method code.

This is not always so. For instance, the following Java code fragment

```
A x;
if (cond)
  x = new B(); // B is a subclass of A
else
  x = new C(); // C is another subclass of A
```

translates to JCVM code where the register `x` acquires type `B` in one arm of the conditional, type `C` in the other arm, and finally type `A` (the l.u.b. of `B` and `C`) when the two arms merge.

Also, an optimizing Java compiler may choose to allocate two source variables whose life spans do not overlap to the same register. Consider for instance the following source code fragment:

```
{ short x; ...  }
{ C y; ...  }
```

The compiler can store `x` and `y` in the same register, since their scopes are disjoint. In the JCVM code, the register will take type `short` in some parts of the method and `C` in others.

In summary, there is no guarantee that the JCVM code given to the verifier will enjoy the two properties mentioned above (operand stack is empty at branch points; registers have only one type

throughout the method), but these two properties hold often enough that it is justified to optimize the bytecode verifier for these two conditions.

One way to proceed from here is to design a data structure for holding the dictionary that is more compact when these two conditions hold. For instance, the "stack is empty" case could be represented specially, and differential encodings could be used to reduce the dictionary size when a register has the same type in many entries.

We decided to take a more radical approach and *require* that all JCVM bytecode accepted by the verifier is such that

- **Requirement R1**: the operand stack is empty at all branch instructions (after popping the branch arguments, if any), and at all branch target instructions (before pushing its results). This guarantees that the operand stack is consistent between the source and the target of any branch (since it is empty at both ends).

- **Requirement R2**: each register has only one type throughout the method code. This guarantees that the types of registers are consistent between source and target of each branch (since they are consistent between any two instructions, actually).

To avoid rejecting correct JCVM code that happens not to satisfy these two requirements, we will rely on a general off-card code transformation that transforms correct JCVM code into equivalent code meeting these two additional requirements. The transformation is described in section 4. We rely on the fact that the violations of requirements R1 and R2 are infrequent to ensure that the code transformations are minor and do not cause a significant increase in code size.

In addition to the two requirements R1 and R2 on verifiable bytecode, we put one additional requirement on the virtual machine:

- **Requirement R3**: on method entry, the virtual machine initializes all registers that are not parameters to the bit pattern representing the `null` object reference.

A method that reads (using the `ALOAD` instruction) from such a register before having stored a valid value in it could obtain an unspecified bit pattern (whatever data happens to be in RAM at the location of the register) and use it as an object reference. This is a serious security threat. The conventional way to avoid this threat is to verify register initialization (no reads before a store) statically, like Sun's bytecode verifier does. To do so, the verifier must then remember the register types at branch target points, which is costly in memory.

The alternate approach we follow here is not to track register initialization during verification, but rely on the virtual machine to initialize non-parameter registers to a safe value: the `null` bit pattern. This way, incorrect code that perform a read before write on a register does not break type safety: all instructions operating on object references test for the `null` reference and raise an exception if appropriate; integer instructions can operate on arbitrary bit patterns without breaking type safety[2].

Clearing registers on method entrance is inexpensive, and it is our understanding that several implementations of the JCVM already do it (even if the specification does not require it) in order to reduce the life-time of sensitive data stored on the stack. In summary, register initialization is a rare example of a type safety property that is easy and inexpensive to ensure dynamically in the virtual machine. Hence, we chose not to ensure it statically by bytecode verification.

[2]A dynamic check must be added to the `RET` instruction, however, so that a `RET` on a register initialized to null will fail instead of jumping blindly to the null code address.

3.2 The algorithm

Given the additional requirements R1, R2 and R3, our bytecode verification algorithm is a simple extension of the algorithm for verifying straight-line code outlined in section 2.1. As previously, the only data structure that we need is *one* stack type and *one* array of types for registers. As previously, the algorithm proceeds by examining in turn every instruction in the method, in code order, and reflecting their effects on the stack and register types. The complete pseudo-code for the algorithm is given in Figure 2. The significant differences with straight-line code verification are as follows.

- When checking a branch instruction, after popping the types of the arguments from the stack, the verifier checks that the stack is empty, and rejects the code otherwise. When checking an instruction that is a branch target, the verifier checks that the stack is empty. (If the instruction is a `JSR` target or the start of an exception handler, it checks that the stack consists of one entry of type "return address" or the exception handler's class, respectively.) This ensures requirement R1.

- When checking a "store" instruction, if τ is the type of the stored value (the top of the stack before the "store"), the type of the register stored into is not replaced by τ, but by the least upper bound of τ and the previous type of the register. This way, register types accumulate the types of all values stored into them, thus progressively determining the unique type of the register as it should apply to the whole method code (requirement R2).

- Since the types of registers can change following the type-checking of a "store" instruction as described above, and therefore invalidate the type-checking of instructions that load and use the stored value, the type-checking of all the instructions in the method body must be repeated until the register types are stable. This is similar to the fixpoint computation in Sun's verifier.

- The dataflow analysis starts, as previously, with an empty stack type and register types corresponding to method parameters set to the types indicated in the method descriptor. Registers not corresponding to parameters are set to $\bot$ (the subtype of all types) instead of $\top$ (the supertype of all types) as a consequence of requirement R3: the virtual machine initializes these registers to the bit pattern representing `null`, and this bit pattern is a correct value of any JCVM type (`short`, `int`, array and reference types, and return addresses) – in other terms, it semantically belongs to the type $\bot$ that is subtype of all other JCVM types. Hence, given requirement R3, it is semantically correct to assign the initial type $\bot$ to registers that are not parameters, like our verification algorithm does.

3.3 Correctness of the verification algorithm

The correctness of our verifier was formally proved using the Coq theorem prover. We developed a mechanically-checked proof that any code that passes our verifier does not cause any run-time type error when executed by a type-level abstract interpretation of a defensive JCVM. To this end, we assume that the verification algorithm succeeded, and extract from its execution an assignment of a stack type and a register type "before" and "after" each instruction in the method. For each instruction, we then prove that starting with an operand stack and registers that match the types

Global variables:

N_r	number of registers
N_s	maximal stack size
$r[N_r]$	array of types for registers
$s[N_s]$	stack type
sp	stack pointer
chg	flag recording whether r changed.

```
Set sp ← 0
Set r[0], ..., r[n − 1] to the types of the method arguments
Set r[n], ..., r[N_r − 1] to ⊥
Set chg ← true

While chg:
  Set chg ← false
  For each instruction i of the method, in code order:

    If i is the target of a branch instruction:
      If sp ≠ 0 and the previous instruction falls through, error
      Set sp ← 0
    If i is the target of a JSR instruction:
      If the previous instruction falls through, error
      Set s[0] ← retaddr and sp ← 1
    If i is a handler for exceptions of class C:
      If the previous instruction falls through, error
      Set s[0] ← C and sp ← 1
    If two or more of the cases above apply, error

    Determine the types a_1, ..., a_n of the arguments of i
    If sp < n, error (stack underflow)
    For k = 1, ..., n:  If s[sp − n + k − 1] is not subtype of a_k, error
    Set sp ← sp − n
    Determine the types r_1, ..., r_m of the results of i
    If sp + m > N_s, error (stack overflow)
    For k = 1, ..., m:  Set s[sp + k − 1] ← r_k
    Set sp ← sp + m
    If i is a store to register number k:
      Determine the type t of the value written to the register
      Set r[k] ← lub(t, r[k])
      If r[k] changed, set chg ← true

    If i is a branch instruction and sp ≠ 0, error

  End for each
End while
Verification succeeds
```

Figure 2: The verification algorithm

"before", a defensive virtual machine can execute the instruction without triggering a run-time type error, and that the operand stack and the registers after the execution of the instruction match the types "after" the instruction inferred by the verifier.

The main difficulty of the proof is to convince the Coq prover that the verification algorithm always terminate, i.e. defines a total function. We do so by proving that the outer `while` loop can only execute a finite number of times, since at each iteration at least one of the entries of the global array of register types increases (is replaced by a strict super-type), and the type lattice has finite height.

3.4 Performance analysis

Our verification algorithm has the same low memory requirements as straight-line code verification: $3S + 3N$ bytes of RAM suffice to hold the stack and register types. In practice, it fits comfortably in 100 bytes of RAM. The memory requirements are independent of the size of the method code, and of the number of branch targets.

Time behavior is similar to that of Sun's algorithm: several passes over the instructions of the method may be required; experimentally, most methods need only two passes (the first determines the types of the registers and the second checks that the fixpoint is reached), and quite a few need only one pass (when all registers are parameters and they keep their initial types throughout the method).

3.5 Subroutines

Subroutines are shared code fragments built from the `JSR` and `RET` instructions and used for compiling the `try...finally` construct in particular [14]. Subroutines complicate Sun-style bytecode verification tremendously. The reason is that a subroutine can be called from different contexts, where registers have different types; checking the type-correctness of subroutine calls therefore requires that the verification of the subroutine code be polymorphic with respect to the types of the registers that the subroutine body does not use [14, section 4.9.6]. This requires a complementary code analysis that identifies the method instructions that belong to subroutines, and match them with the corresponding `JSR` and `RET` instructions. During verification, the results of this analysis are used to type-check `JSR` and `RET` instructions in a polymorphic way. See [26, 23, 25] for formalizations of this approach. Alternate approaches are described in [11, 19, 13].

All these complications (and potential security holes) disappear in our bytecode verification algorithm: since it ensures that a register has the same type throughout the method code, it ensures that the whole method code, including subroutines, is monomorphic with respect to the types of all registers. Hence, there is no need to verify the `JSR` and `RET` instructions in a special, polymorphic way: `JSR` is treated as a regular branch that also pushes a value of type "return address" on the stack; and `RET` is treated as a branch that can go to any instruction that follows a `JSR` in the current method. No complementary analysis of the subroutine structure is required, and it suffices to have one type constant `retaddr` to represent return addresses, instead of `retaddr` types annotated with code locations as in [26], or with usage bit vectors as in [14].

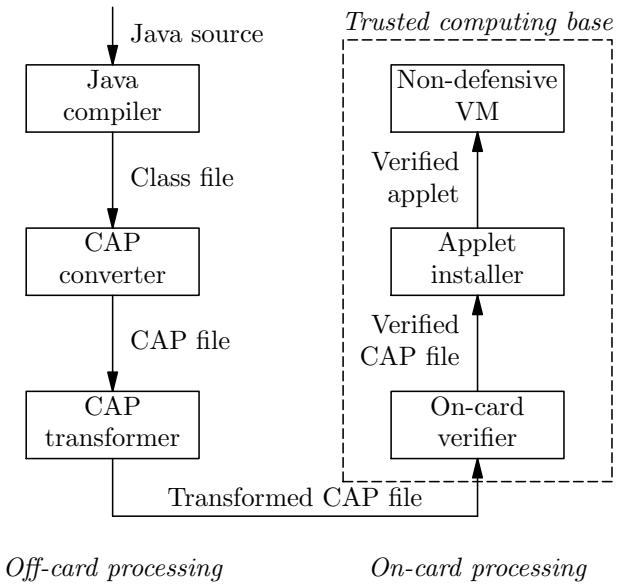

Figure 3: Architecture of the system

4 OFF-CARD CODE TRANSFORMATIONS

As explained in section 3.1, our on-card verifier accepts only a subset of all type-correct applets: those whose code satisfies the two additional requirements R1 (operand stack is empty at branch points) and R2 (registers have unique types). To ensure that all correct applets pass verification, we could compile them with a special Java compiler that generates JVM bytecode satisfying requirements R1 and R2, for instance by expanding conditional expressions e_1 `?` e_2 `:` e_3 into `if...then...else` statements, and by assigning distinct register to each source-level local variable.

Instead, we found it easier and more flexible to let applet developers use a standard Java compiler and Java Card converter of their choice, and perform an off-card code transformation on the compiled code to produce an equivalent compiled code that satisfies the additional requirements R1 and R2 and can therefore pass the on-card verifier (see Figure 3).

Two main transformations are performed: stack normalization (to ensure that the operand stack is empty at branch points) and register reallocation (to ensure that a given register is used with only one type). Both transformations are performed method by method, and are type-directed: they operate on method code annotated by the stack type and types of registers at each instruction. This type information is obtained by a preliminary pass of bytecode verification using Sun's algorithm. (This off-card verification, intended to support transformations of the code, is not to be confused with the on-card verification, intended to establish its type correctness; only the latter is part of the trusted computing base.)

4.1 Stack normalization

The idea underlying stack normalization is quite simple: whenever the original code contains a branch with a non-empty stack, we insert stores to fresh registers before the branch, and loads from the same registers at the branch target. This effectively empties the operand stack into the

fresh registers before the branch, and restore it to its initial state after the branch. Consider for example the following Java statement: `C.m(b ? x : y);`. It compiles down to the JCVM code fragment shown below on the left.

```
        sload Rb
        ifeq lbl1
        sload Rx
        goto lbl2
lbl1:   sload Ry
lbl2:   invokestatic C.m
```

```
        sload Rb
        ifeq lbl1
        sload Rx
        sstore Rtmp
        goto lbl2
lbl1:   sload Ry
        sstore Rtmp
lbl2:   sload Rtmp
        invokestatic C.m
```

Here, `Rx`, `Ry` and `Rb` are the numbers for the registers holding `x`, `y` and `b`. The result of type inference for this code indicates that the stack is non-empty across the `goto` to `lbl2`: it contains one entry of type `short`. Stack normalization therefore rewrites it into the code shown above on the right, where `Rtmp` is the number of a fresh, unused register. The `sstore Rtmp` before `goto lbl2` empties the stack, and the `sload Rtmp` at `lbl2` restore it before proceeding with the `invokestatic`. Since the `sload Ry` at `lbl1` falls through the instruction at `lbl2`, we must treat it as an implicit jump to `lbl2` and also insert a `sstore Rtmp` between the `sload Ry` and the instruction at `lbl2`.

(Allocating fresh temporary registers such as `Rtmp` for each branch target needing normalization may seem wasteful. Register reallocation, as described in section 4.2, is able to "pack" these variables, along with the original registers of the method code, thus minimizing the number of registers really required.)

The actual stack normalization transformation is slightly more complex, due to branch instructions that pop arguments off the stack, and also to the fact that a branch instruction needing normalization can be itself the target of another branch instruction needing normalization.

Stack normalization starts by detecting every instruction i that is targets of branches and where the operand stack before the execution of the instruction is not empty, as shown by the stack type annotating the instruction. Let $n > 0$ be the height of the operand stack in words. We generate n fresh registers $l_1 \ldots, l_n$ and associate them to i.

In a second pass, each instruction i of the method is examined in turn.

- If the instruction i is a branch target with a non-empty operand stack:

  Let $l_1 \ldots, l_n$ be the fresh registers previously associated with i.

  - If the instruction before i does not fall through (i.e. it is an unconditional branch, a return or a `throw`), insert loads from $l_1 \ldots, l_n$ before i and redirect the branches to i so that they branch to the first load thus inserted:

$$\texttt{lbl:}\ i \quad \longrightarrow \quad \begin{array}{ll} \texttt{lbl:} & x\texttt{load}\ l_1 \\ & \ldots \\ & x\texttt{load}\ l_n \\ & i \end{array}$$

– If the instruction before i falls through, insert stores to $l_n, \ldots, l_1$, then loads from $l_1 \ldots, l_n$, before i, and redirect the branches to i so that they branch to the first load thus inserted:

$$\begin{array}{rcl} \texttt{lbl:}\ i & \longrightarrow & x\texttt{store}\ l_n \\ & & \ldots \\ & & x\texttt{store}\ l_1 \\ & & \texttt{lbl:}\ x\texttt{load}\ l_1 \\ & & \ldots \\ & & x\texttt{load}\ l_n \\ & & i \end{array}$$

- If the instruction i is a branch to instruction j and the operand stack is not empty at j:

  Let $l_1 \ldots, l_n$ be the fresh registers previously associated with j. Let k be the number of arguments popped off the stack by the branch instruction i. (This can be 0 for a simple `goto`, 1 for multi-way branches, and 1 or 2 for conditional branches.)

  – If the instruction i does not fall through (unconditional branch), insert before i code to swap the top k words of the stack with the n words below, followed by stores to $l_n, \ldots, l_1$:

$$\begin{array}{rcl} i & \longrightarrow & \texttt{swap\_x}\ k, n \\ & & x\texttt{store}\ l_n \\ & & \ldots \\ & & x\texttt{store}\ l_1 \\ & & i \end{array}$$

  – If the instruction i can fall through (conditional branch), do as in the previous case, then insert after i loads from $l_1 \ldots, l_n$:

$$\begin{array}{rcl} i & \longrightarrow & \texttt{swap\_x}\ k, n \\ & & x\texttt{store}\ l_n \\ & & \ldots \\ & & x\texttt{store}\ l_1 \\ & & i \\ & & x\texttt{load}\ l_1 \\ & & \ldots \\ & & x\texttt{load}\ l_n \end{array}$$

- In the rare case where the instruction i is both a branch target with a non-empty stack *and* a branch to a target j with a non-empty stack, we combine the two transformations above. For a worst-case example, assume that the instruction before i falls through and i itself falls through. Let $l_1 \ldots, l_n$ be the fresh registers associated with i, and $t_1, \ldots, t_p$ those associated with j. Let k be the number of arguments popped off the stack by the branch instruction i. The transformation is then as follows:

```
lbl: i   ⟶          xstore l_n
                    ...
                    xstore l_1
               lbl: xload l_1
                    ...
                    xload l_n
                    swap_x k,n
                    xstore t_p
                    ...
                    xstore t_1
                    i
                    xload t_1
                    ...
                    xload t_p
```

Since the transformations above are potentially costly in terms of code size and number of registers, we first apply standard "tunneling" optimizations to the original code: replace branches to `goto lbl` by a direct branch to `lbl`; replace unconditional branches to a `return` or `athrow` instruction by a copy of the `return` or `athrow` instruction itself. This reduces the number of branches, hence the number of branches that require stack normalization. For instance, the common Java idiom

```
return e1 ? e2 : e3;
```

is usually compiled to the following code

```
        evaluate e1
        ifeq lbl1
        evaluate e2
        goto lbl2
lbl1:   evaluate e2
lbl2:   sreturn
```

This code needs a stack normalization at `goto lbl2` and at `lbl2` itself. The tunneling optimization replaces `goto lbl2` by a direct `sreturn`:

```
        evaluate e1
        ifeq lbl1
        evaluate e2
        sreturn
lbl1:   evaluate e2
        sreturn
```

and this code requires no stack normalization, since it already conforms to requirement R1.

4.2 Register reallocation

The second code transformation performed off-card consists in re-allocating registers (i.e. change the register numbers) in order to ensure requirement R2: a register is used with only one type

throughout the method code. This can always be achieved by "splitting" registers used with several types into several distinct registers, one per use type. However, this can increase markedly the number of registers required by a method.

Instead, we use a more sophisticated register reallocation algorithm, derived from the well-known algorithms for global register allocation via graph coloring [4, 2]. This algorithm tries to reduce the number of registers by reusing the same register as much as possible, i.e. to hold source variables that are not live simultaneously and that have the same type. Consequently, it is very effective at reducing inefficiencies in the handling of registers, either introduced by the stack normalization transformation, or left by the Java compiler.

Consider the following example (original code on the left, result of register reallocation on the right).

```
sconst_1                    sconst_1
sstore 1                    sstore 1
sload 1                     sload 1
sconst_2                    sconst_2
sadd                        sadd
sstore 2                    sstore 1
new C                       new C
astore 1                    astore 2
...                         ...
```

In the original code, register 1 is used with two types: first to hold values of type `short`, then to hold values of type `C`. In the transformed code, these two roles of register 1 are split into two distinct registers, 1 for the `short` role and 2 for the `C` role. In parallel, the reallocation algorithm notices that, in the original code, register 2 and the `short` role of register 1 have disjoint live ranges and have the same type. Hence, these two registers are merged into register 1 in the transformed code. The end result is that the number of registers stays constant.

The register reallocation algorithm is essentially identical to Briggs' variant of Chaitin's graph coloring allocator [4, 2], with additional type constraints reflecting requirement R2.

- Compute live ranges for every register in the method code as described in [16, section 16.3].
- (New step.) Compute the principal type for every live range. This is the least upper bound of the types of all values stored in the corresponding register by `store` instructions belonging to the live range.
- Build the interference graph between live ranges [1, section 9.7]. The nodes of this undirected graph are the live ranges, and there is an edge between two live ranges if and only if they interfere, i.e. one contains a `store` instruction on the register associated with the other.
- (New step.) Reflect requirement R2 in the interference graph by adding interference edges between any two live ranges that do not have the same principal type.
- Coalescing: detect register-to-register copies, i.e. sequences of the form `load` i`; store` j, such that the source i and the destination j do not interfere; coalesce the two live ranges associated with i and j, treating them as a single register, and remove the copy instructions. This is essentially Chaitin's aggressive coalescing strategy [4].

- Color the inference graph: assign a new register number to every live range in such a way that two interfering live ranges have distinct register numbers. Try to minimize the number of "colors" (i.e. registers) used. Although optimal graph coloring is NP-complete, there exists linear-time algorithms that give quite good results on coloring problems corresponding to register allocation. We used the algorithm described in [2], with the obvious simplification that we never need to "spill" registers on the stack, since in our case the number of registers is not bounded in advance by the hardware.

The reallocation algorithm in general and the coalescing pass in particular are very effective at reducing inefficiencies in the handling of registers, either introduced by the stack normalization transformation, or by the Java compiler. Consider for instance the following Java code

```
short s = b ? x : y;
```

After compilation and stack normalization, we obtain the following JCVM code:

```
      sload Rb
      ifeq lbl1
      sload Rx
      sstore Rtmp
      goto lbl2
lbl1: sload Ry
      sstore Rtmp
lbl2: sload Rtmp
      sstore Rs
```

The `sload Rtmp; sstore Rs` is coalesced since `Rtmp` and `Rs` do not interfere, resulting in more efficient code:

```
      sload Rb
      ifeq lbl1
      sload Rx
      sstore Rs
      goto lbl2
lbl1: sload Ry
      sstore Rs
lbl2:
```

that corresponds to the Java source code

```
short s; if (b) { s = x; } else { s = y; }
```

5 EXPERIMENTAL RESULTS

5.1 Off-card transformation

Table 1 shows results obtained by transforming 8 packages from Sun's Java Card development kit and from Gemplus' Pacap test applet. The Java compiler used is `javac` from JDK 1.2.2.

Package	Code size (bytes)			Registers
	Orig.	Transf.	Incr.	used
`java.lang`	92	91	−1.0%	0.0%
`javacard.framework`	4047	4142	+2.3%	+0.3%
`com.sun.javacard.HelloWorld`	100	99	−1.0%	0.0%
`com.sun.javacard.JavaPurse`	2558	2531	−1.0%	−8.3%
`com.sun.javacard.JavaLoyalty`	207	203	−1.9%	0.0%
`com.sun.javacard.installer`	7043	7156	+1.6%	−7.5%
`com.gemplus.pacap.utils`	1317	1258	−4.4%	−13.5%
`com.gemplus.pacap.purse`	19813	19659	−0.7%	−6.9%
Total	35177	35139	−0.1%	−4.5%

Table 1: Effect of the off-card code transformation on code size and register requirements

The effect of the transformation on the code size is almost negligible. In the worst case (package `javacard.framework`), the code size increases by 2.3%. On several packages, the code size actually decreases by as much as 4.4% due to the clean-up optimisations reducing inefficiencies left by the Java compiler. Similarly, the requirements in registers globally decreases by about 4%.

To test a larger body of code, we used a version of the off-card transformer that works over Java class files (instead of Java Card CAP files) and transformed all the classes from the Java Runtime Environment version 1.2.2, that is, about 1.5 Mbyte of JVM code. The results are very similar: globally, code size increases by 0.7%; register needs decrease by 1.3%.

Figure 4 shows the increase in code size for each of the concrete methods of the packages studied. Each point represent one method, with the original size of the method code s (in bytes) in abscissa, and the code size increase factor $(s' - s)/s$ in ordinate, where s' is the size of the method code after transformation. The dots are heavily clustered around the horizontal axis. For the Java Card packages, approximately 350 methods are displayed, and only 15 show a code size increase above 10%, with one relatively small method suffering a 75% increase. Large relative variations in code size occur only for small methods (50 bytes or less); larger methods exhibit smaller variations, which explain why the total code size increases only by 0.9%. The Java Runtime Environment, totalling approximately 26000 methods, exhibits a similar behavior: a handful of small methods suffer a code size increase above 100%, but almost all methods are clustered along the horizontal axis, especially the larger methods.

5.2 On-card verifier

We present here preliminary results obtained on an implementation of our bytecode verifier running on a Linux PC. A proper on-card implementation is in progres at one of our licensees, but we are not in a position to give precise results concerning this implementation.

Concerning the size of the verifier, the bytecode verification algorithm, implemented in ANSI C, compiles down to 11 kilobytes of Intel IA32 code, and 9 kilobytes of Atmel AVR code. A proof-of-concept reimplementation in hand-written ST7 assembly code fits in 4.5 kilobytes of code.

Concerning verification speed, the PC implementation of the verifier, running on a 500 MHz Pentium III, takes approximately 1.5 ms per kilobyte of bytecode. On a typical 8051-style smartcard

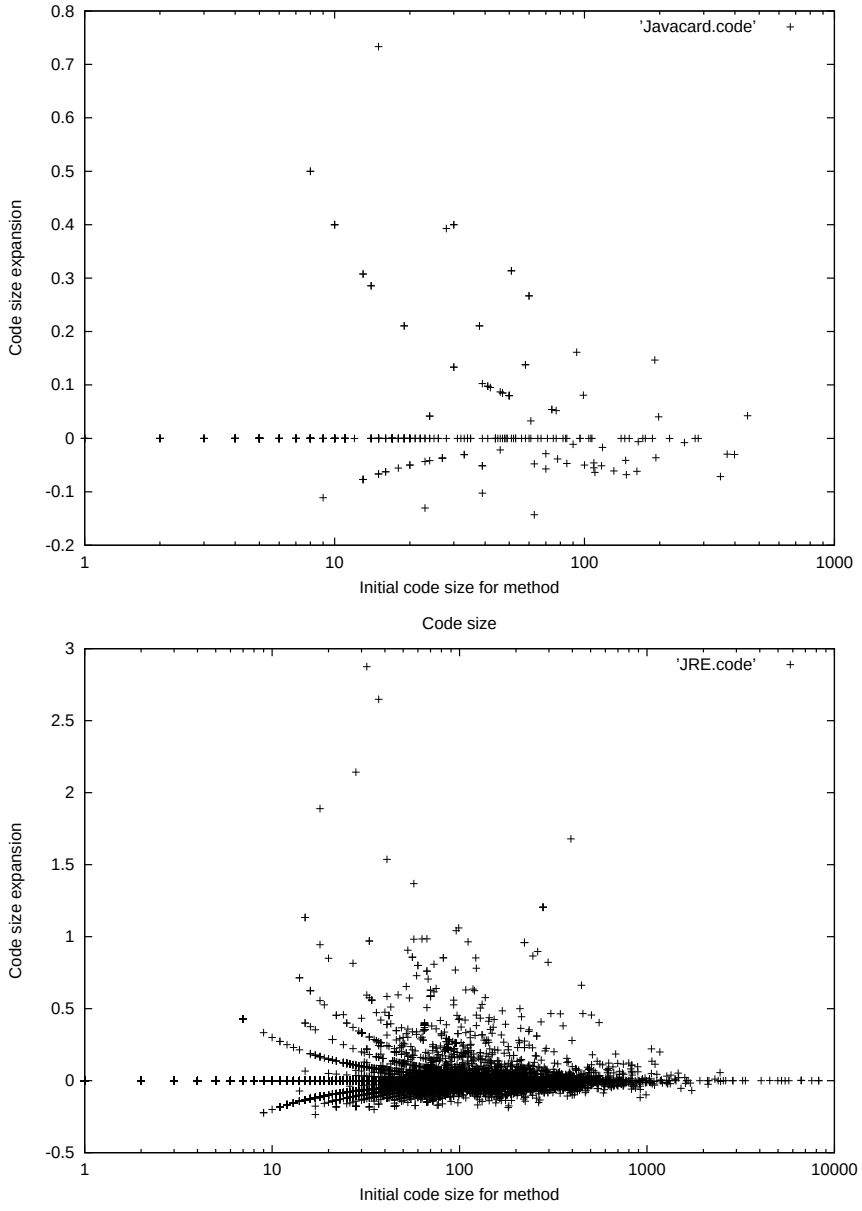

Figure 4: Relative increase in code size as a function of the original method code size (in bytes). Top: Java Card packages; bottom: Java Runtime Environment packages.

processor, an on-card implementation takes approximately 1 second per kilobyte of bytecode, or about 2 seconds to verify an applet the size of `JavaPurse`. Notice that the verifier performs no EEPROM writes and no communications, hence its speed benefits linearly from higher clock rates or more efficient processor cores.

Concerning the number of iterations required to reach the fixpoint in the bytecode verification algorithm, the first 6 packages we studied contain 7077 JCVM instructions and require 11492 calls to the function that analyzes individual instructions. This indicates that each instruction is analyzed 1.6 times on average before reaching the fixpoint. This figure is surprisingly low; it shows that a "perfect" verification algorithm that analyzes each instruction exactly once, such as [24], would only be 38% faster than ours.

6 RELATED WORK

6.1 Lightweight bytecode verification

The work most closely related to ours is the lightweight bytecode verification of Rose and Rose [24], also found in Sun's KVM/CLDC architecture [27] and in the Facade project [10]. Inspired by proof-carrying code [17], lightweight bytecode verification consists in sending, along with the code to be verified, pre-computed stack and register types for each branch target. These pre-computed types are called "certificates" or "stack maps". Verification then simply checks the correctness of these types, using a simple variant of straight-line verification, instead of inferring them by fixpoint iteration, as in Sun's verifier.

The interest for an on-card verifier is twofold. The first is that fixpoint iteration is avoided, thus making the verifier faster. (As mentioned at the end of section 5.2, the performance gain thus obtained is modest.) The second is that the certificates can be stored temporarily in EEPROM, since they do not need to be updated repeatedly during verification. The RAM requirements of the verifier become similar to those of our verifier: only the current stack type and register type need to be kept in RAM.

There are two issues with Rose and Rose's lightweight bytecode verification. A minor issue is that it currently does not deal with subroutines, more specifically with polymorphic typing of subroutines as described in section 3.5. To work around this issue, the KVM implementation of lightweight bytecode verification simply expands all subroutines at point of call during the off-card generation of certificates. Current Java compilers use subroutines sparingly in the code they generate, so the impact of this expansion on code size is negligible. However, reducing code size is important in the Java Card world, and space-reducing compilers or post-optimizers could make more intensive use of subroutines as a code sharing device.

A more serious issue is the size of the certificates that accompany the code. Table 2 shows, for each of our test packages, the size of the certificates generated by the `preverify` tool from Sun's KVM/CLDC environment. On average, the size of the certificates is 50% of the size of the code they annotate. The format of certificates generated by `preverify` is relatively compact (1 byte for base types, 3 bytes for class types); further compression is certainly possible, but our experiments indicate that it is difficult to go below 20% of the code size. Hence, significant free space in EEPROM is required for storing temporarily the certificates during the verification of large packages, and this can be a serious practical issue in the context of Java Card. In contrast, our verification technology only requires at most 2% of extra EEPROM space.

Package	Code size	Certificate size	Relative size
`javacard.framework`	4047	1854	46%
`com.sun.javacard.HelloWorld`	100	36	36%
`com.sun.javacard.JavaPurse`	2558	1949	76%
`com.sun.javacard.JavaLoyalty`	207	218	105%
`com.sun.javacard.installer`	7043	3520	50%
`com.gemplus.pacap.utils`	1317	1013	77%
`com.gemplus.pacap.purse`	19813	8835	44%
Total	35177	17425	50%

Table 2: Size of certificates in the lightweight bytecode verification approach

6.2 Formalizations of Sun's verifier

Challenged by the lack of precision in the reference publications of Sun's verifier [9, 31, 14], many researchers have published rational reconstructions, formalizations, and formal proofs of correctness of various subsets of Sun's verifier [6, 22, 21, 23, 7, 18, 25]. (See Hartel and Moreau's survey [12] for a more detailed description.) These works were influential in understanding the issues, uncovering bugs in Sun's implementation of the verifier, and generating confidence in the algorithm. Unfortunately, most of these works address only a subset of the verifier. In particular, [25] is the only published proof of the correctness of Sun's polymorphic typing of subroutines in the presence of exceptions.

6.3 Other approaches to bytecode verification

A different approach to bytecode verification was proposed by Posegga [20] and further refined by Brisset [3]. This approach is based on model checking of a type-level abstract interpretation of a defensive Java virtual machine. It trivializes the problem with polymorphic subroutines and exceptions, but is very expensive (time and space exponential in the size of the method code), thus is not suited to on-card implementation. Leroy [13] describes a less expensive variant of this approach, based on polyvariant verification of subroutines.

7 CONCLUSIONS

The approach described in this article – off-card code transformations to simplify the bytecode verification process – leads to a novel bytecode verification algorithm that is perfectly suited to on-card implementation, due to its low RAM requirements. It is superior to Rose and Rose's lightweight bytecode verification in that it does not force subroutines to be expanded beforehand, and requires much less additional EEPROM space (2% of the code size vs. 50% for lightweight bytecode verification).

On-card bytecode verification is the missing link in the Java Card vision of multi-application smart cards with secure, efficient post-issuance downloading of applets. We believe that our bytecode verifier is a crucial enabling technology for making this vision a reality.

References

[1] Aho AV, Sethi R, Ullman JD. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] Briggs P, Cooper KD, Torczon L. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.* 1994; **16**(3):428–455.

[3] Brisset P. Vers un vérifieur de bytecode Java certifié. Seminar given at École Normale Supérieure, Paris, October 2nd 1998.

[4] Chaitin GJ. Register allocation and spilling via graph coloring. *SIGPLAN Notices* 1982, **17**(6):98–105.

[5] Chen Z. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.

[6] Cohen R. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997.

[7] Freund SN, Mitchell JC. The type system for object initialization in the Java bytecode language. *ACM Trans. Prog. Lang. Syst.* 1999, **21**(6):1196–1250.

[8] Gong L. *Inside Java 2 platform security: architecture, API design, and implementation*. The Java Series. Addison-Wesley, 1999.

[9] Gosling JA. Java intermediate bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118. ACM, 1995.

[10] Grimaud G, Lanet J-L, Vandewalle J-J. FACADE – a typed intermediate language dedicated to smart cards. In *Software Engineering - ESEC/FSE '99*, volume 1687 of *LNCS*, pages 476–493. Springer-Verlag, 1999.

[11] Hagiya M, Tozawa A. On a new method for dataflow analysis of Java virtual machine subroutines. In *SAS'98*, Levi G (ed.), *LNCS* 1503, pages 17–32. Springer-Verlag, 1998.

[12] Hartel PH and Moreau LAV. Formalizing the safety of Java, the Java virtual machine and Java card. *ACM Computing Surveys*, 2001. To appear.

[13] Leroy X. Java bytecode verification: an overview. In *Computer Aided Verification, CAV 2001*, Berry G, Comon H, Finkel A (eds.), *LNCS* 2102, pages 265–285. Springer-Verlag, 2001.

[14] Lindholm T, Yellin F. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.

[15] McGraw G, Felten E. *Securing Java*. John Wiley & Sons, 1999.

[16] Muchnick SS. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[17] Necula GC. Proof-carrying code. In *24th symp. Principles of Progr. Lang*, pages 106–119. ACM Press, 1997.

[18] Nipkow T. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS'01)*, *LNCS* 2030, pages 347–363. Springer-Verlag, 2001.

[19] O'Callahan R. A simple, comprehensive type system for Java bytecode subroutines. In *26th symp. Principles of Progr. Lang*, pages 70–78. ACM Press, 1999.

[20] Posegga J, Vogt H. Java bytecode verification using model checking. In *Workshop Fundamental Underpinnings of Java*, 1998.

[21] Pusch C. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *TACAS'99*, Cleaveland WR (ed.), volume 1579 of *LNCS*, pages 89–103. Springer-Verlag, 1999.

[22] Qian Z. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In *Formal syntax and semantics of Java*, Alves-Foss J (ed.), *LNCS* 1523. Springer-Verlag, 1998.

[23] Qian Z. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Prog. Lang. Syst.* 2000; **22**(4):638–672.

[24] Rose E, Rose K. Lightweight bytecode verification. In *Workshop Fundamental Underpinnings of Java*, 1998.

[25] Stärk R, Schmid J, Börger E. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[26] Stata R, Abadi M. A type system for Java bytecode subroutines. *ACM Trans. Prog. Lang. Syst.* 1999; **21**(1):90–137.

[27] Sun Microsystems. Java 2 platform micro edition technology for creating mobile devices. White paper, http://java.sun.com/products/cldc/wp/KVMwp.pdf, 2000.

[28] Sun Microsystems. JavaCard 2.1.1 runtime environment specification, 2000.

[29] Sun Microsystems. JavaCard 2.1.1 virtual machine specification, 2000.

[30] Vigna G (ed.). *Mobile Agents and Security*, *LNCS* 1419. Springer-Verlag, 1998.

[31] Yellin F. Low level security in Java. In *Proc. 4th International World Wide Web Conference*, pages 369–379. O'Reilly, 1995.