On-card Bytecode Verification for Java Card

Xavier Leroy

On-card Bytecode Verification for Java Card

Xavier Leroy

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

Abstract. This paper presents a novel approach to the problem of bytecode verification for Java Card applets. Owing to its low memory requirements, our verification algorithm is the first that can be embedded on a smart card, thus increasing tremendously the security of post-issuance downloading of applets on Java Cards.

1 Introduction

The Java Card architecture for smart cards [4] bring two major innovations to the smart card world: first, Java cards can run multiple applications, which can communicate through shared objects; second, new applications, called *applets*, can be downloaded on the card post issuance. These two 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 [23, 11]. 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 [7]. The security of the sandbox model relies on the following three components:

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.

* This work was performed when the author was full-time at Trusted Logic. He is currently affiliated with INRIA Rocquencourt (domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France) and part-time consultant for Trusted Logic.

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 [8, 24, 10].

The Java Card architecture features components 1 and 2 of the sandbox model: applets are executed by the Java Card virtual machine [22], and the Java Card runtime environment [21] 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. 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 unfeasible. In this paper, 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. Along with auxiliary consistency checks on the CAP file structure, not described in this paper for lack of space, the bytecode verifier described in this paper 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 paper 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 [8, 24, 10].

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 those found in 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 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 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 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.

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. Hence, it must propagate the inferred stack and register types to all possible successors of the forking instruction. Joins are even harder: an instruction 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 that smallest type that is assignment-compatible with the two types.) 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 [10, section 4.9] for a more detailed description.

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, 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.

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 paper follows from a careful analysis of the shortcomings of Sun's algorithm, namely that a copy of the stack type and register 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 stack only for evaluating expressions, but never store the values of Java local variables on the stack. Consequently, the 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, the branches generated when compiling these constructs naturally occur in the context of an empty stack. The only exception is the conditional expression e_1 `?` e_2 `:` e_3, which indeed generates a branch on a non-empty stack. As regards to registers, Java compilers very often allocate a distinct JCVM register for each local variable in the Java source. This register is naturally used with only one type, that of the declaration of the local variable.

Of course, there is no guarantee that the JCVM code given to the verifier will enjoy the two properties mentioned above (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 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 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.

3.2 The Algorithm

Given the two additional requirements R1 and R2, 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 Fig. 1. 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. Locals not corresponding to parameters are set to $\bot$ (the subtype of all types) instead of $\top$ (the supertype of all types) for reasons that are explained in section 3.4 below.

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 arg.
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 n:
      Determine the type t of the value written to the register
      Set r[n] ← lub(t, r[n])
      If r[n] changed, set chg ← true

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

  End for each
End while
Verification succeeds
```

Fig. 1. The verification algorithm

The correctness of our verifier was formally proved using the Coq theorem prover. More precisely, we developed a mechanically-checked proof that any code that passes our verifier does not cause any run-time type error when run through a type-level abstract interpretation of a defensive JCVM.

3.3 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 type throughout the method).

3.4 Initialization of Registers

Unlike Sun's, our verification algorithm cannot guarantee that registers are initialized (stored into) before use. The reason is that since we have only one set of register types for the whole method, we cannot analyze precisely the situation where a register is initialized on one branch of a conditional and not on the other branch.

The JVM and JCVM specifications do not require the virtual machine to initialize non-parameter registers on entry to a method. Hence, 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.

There are two ways to avoid this threat. One is to verify register initialization (no reads before a store) statically, as part of the bytecode verifier. The other is to rely on the virtual machine to initialize, on entry to a method, all registers that are not method parameters to the bit-pattern representing the `null` object reference. 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. (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.)

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.

Since the bit pattern representing `null` is a correct value of any JCVM type (`short`, `int`, array and reference types, and return addresses), it semantically belongs to the type $\perp$ that is subtype of all other JCVM types. Hence, assuming initialization to `null` in the virtual machine, it is semantically correct to assign the initial type $\perp$ to registers that are not parameters, like our verification algorithm does.

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 [10]. 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 [10, 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. See [19, 17] for formalizations of this approach.

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.

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 (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 JavaCard converter of their choice, and perform an off-card code transformation on the compiled code to produce an equivalent

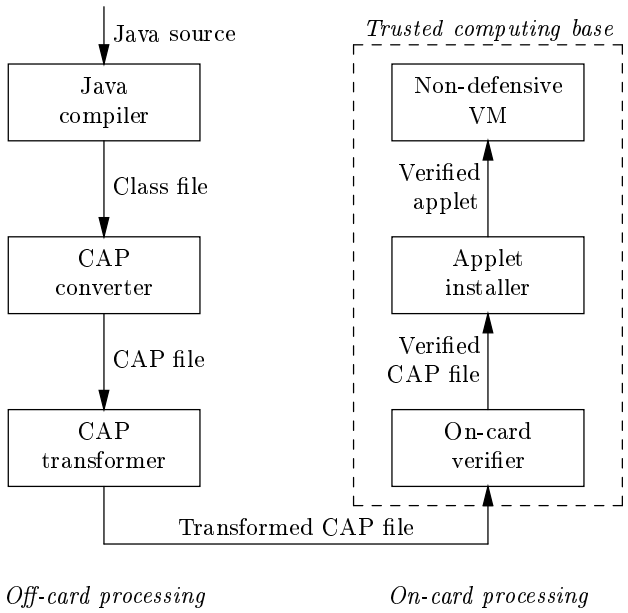

Fig. 2. Architecture of the system

compiled code that satisfies the additional requirements R1 and R2 and can therefore pass the on-card verifier (see Fig. 2).

Two main transformations are performed: stack normalization (to ensure that the stack is empty at branch points) and register reallocation (to ensure that a given register is used with only one type).

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 stack into the fresh registers before the branch, and restore the stack 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 Lb
        ifeq lbl1
        sload Lx
        goto lbl2
lbl1:   sload Ly
lbl2:   invokestatic C.m
```

```
        sload Lb
        ifeq lbl1
        sload Lx
        sstore Ltmp
        goto lbl2
lbl1:   sload Ly
        sstore Ltmp
```

```
                                    lbl2: sload Ltmp
                                          invokestatic C.m
```

Here, `Lx`, `Ly` and `Lb` 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 `Ltmp` is the number of a fresh, unused register. The `sstore Ltmp` before `goto lbl2` empties the stack, and the `sload Ltmp` at `lbl2` restore it before proceeding with the `invokestatic`. Since the `sload Ly` at `lbl1` falls through the instruction at `lbl2`, we must treat it as an implicit jump to `lbl2` and also insert a `sstore Ltmp` between the `sload Ly` and the instruction at `lbl2`.

(Allocating fresh temporary registers such as `Ltmp` 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.)

By lack of space, we omit a detailed presentation of the actual stack normalization transformation. It follows the approach outlined above, with some extra complications 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.

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. 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 [3, 1], with additional type constraints reflecting requirement R2. More precisely, we add edges in the interference graph between live ranges that do not have the same principal type, thus guaranteeing that they will be assigned different registers.

5 Experimental Results

5.1 Off-card Transformation

The table below shows results obtained by transforming 6 packages from Sun's Java Card development kit.

Package	Code size (bytes)			Resident size (bytes)			Registers
	Orig.	Transf.	Incr.	Orig.	Transf.	Incr.	
`java.lang`	92	91	-1%	320	319	-0.3%	0.0%
`javacard.framework`	4047	4142	+2.3%	5393	5488	+1.8%	+0.3%
`com.sun.javacard.HelloWorld`	100	99	-1%	220	219	-0.5%	0.0%
`com.sun.javacard.JavaPurse`	2558	2531	-1%	3045	3018	-0.8%	-8.3%
`com.sun.javacard.JavaLoyalty`	207	203	-1.9%	365	361	-1%	0.0%
`com.sun.javacard.installer`	7043	7156	+1.6%	8625	8738	+1.3%	-7.5%
Total	14047	14222	+1.2%	17968	18143	+0.9%	-4.2%

The code size increase caused by the transformation is almost negligible: the size of the `Method` component increases by 1.2%; the resident size (total size of all components that remain on the card after installation) increases by 0.9%. 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: code size increases by 0.7%; registers decrease by 1.3%.

The transformer performs clean-up optimizations (branch tunneling, register coalescing) whose purpose is to reduce inefficiencies introduced by other transformations. These optimizations are also quite effective at reducing inefficiencies left by the Java compiler, resulting in code size decreases of up to 1.9% for some packages. Similarly, the packing of registers actually reduces the maximal number of registers in most packages.

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 progress, but we are not in a position to give results concerning this implementation.

Bytecode verification proper (ensuring that method code is type-safe), written 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.

In addition to verifying the bytecode of methods, our implementation also checks the structural consistency of CAP file components. Since the CAP file format is extremely complex [22, chapter 6], CAP file consistency checking takes a whopping 12 kilobytes of Intel IA32 code. However, when integrating the verifier with an actual Java Card VM, many of these consistency checks become redundant with checks already performed by the VM or the installer, or useless because they apply to CAP file information that the VM ignores. Programming tricks such as table-driven automata can also be used to reduce further the code size of consistency checking, at some expense in execution speed.

The PC implementation of the verifier, running on a 500 Mhz Pentium III, takes approximately 1.5 ms per kilobyte of bytecode. Extrapolating this figure to a typical 8-byte smartcard processor (e.g. 8051 at 5 Mhz), we estimate that an on-card implementation should take less than 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 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 [18], would only be 38% faster than ours.

6 Related Work

The work most closely related to ours is the lightweight bytecode verification of Rose and Rose [18], also found in the KVM architecture [20] and in [9]. Inspired by proof-carrying code [12], lightweight bytecode verification consists in sending, along with the code to be verified, pre-computed stack and register types for each branch target. Verification then simply checks the correctness of these pre-computed 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 stack and register types at branch targets 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 problems with Rose and Rose's lightweight bytecode verification. One is that it currently does not deal with subroutines, more specifically with polymorphic typing of subroutines as described in section 3.5. Subroutines are part of the JCVM specification, and could be useful as a general code sharing device for reducing bytecode size. The second issue is the size of the "certificate", that is, the pre-computed stack and register types that accompany the code. Our experiments indicate that, using a straightforward representation, certificates are about the same size as the code. Even with a more complex, compressed representation, certificates are still 20% of the code size. Hence, significant free space in EEPROM is required for storing temporarily the certificates during the verification of large packages. In contrast, our verification technology only requires at most 1–2% of extra EEPROM space.

Challenged by the lack of precision in the reference publications of Sun's verifier [8, 24, 10], many researchers have published rational reconstructions, formalizations, and formal proofs of correctness of various subsets of Sun's verifier [5, 16, 15, 17, 6, 13]. 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, none of them proves the correctness of Sun's polymorphic typing of subroutines in the presence of exceptions.

A different approach to bytecode verification was proposed by Posegga [14] and further refined by Brisset [2]. 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.

7 Conclusions

The novel bytecode verification algorithm described in this paper 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 handles subroutines, and requires much less additional EEPROM space (1–2% of the code size vs. 20–100% for lightweight bytecode verification).

On-card bytecode verification is the missing link in the Javacard 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. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.*, 16(3):428–455, 1994.
2. P. Brisset. Vers un vérifieur de bytecode Java certifié. Seminar given at Ecole Normale Supérieure, Paris, Oct 2nd 1998.
3. G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, 1982.
4. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.
5. R. Cohen. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997.
6. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Trans. Prog. Lang. Syst.*, 22(5), 2000.
7. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. The Java Series. Addison-Wesley, 1999.
8. J. A. Gosling. Java intermediate bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118. ACM, 1995.
9. G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. 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.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.
11. G. McGraw and E. Felten. *Securing Java*. John Wiley & Sons, 1999.
12. G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM Press, 1997.
13. T. Nipkow. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS'01)*. Springer-Verlag, 2001. To appear.
14. J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop Fundamental Underpinnings of Java*, 1998.
15. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. R. Cleaveland, editor, *TACAS'99*, volume 1579 of *LNCS*, pages 89–103. Springer-Verlag, 1999.
16. Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal syntax and semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.
17. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Prog. Lang. Syst.*, 22(4):638–672, 2000.
18. E. Rose and K. Rose. Lightweight bytecode verification. In *Workshop Fundamental Underpinnings of Java*, 1998.
19. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Trans. Prog. Lang. Syst.*, 21(1):90–137, 1999.
20. Sun Microsystems. Java 2 platform micro edition technology for creating mobile devices. White paper, 2000.
21. Sun Microsystems. Java Card 2.1.1 runtime environment specification, 2000.
22. Sun Microsystems. Java Card 2.1.1 virtual machine specification, 2000.
23. G. Vigna, editor. *Mobile Agents and Security*. Number 1419 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
24. F. Yellin. Low level security in Java. In *Proc. 4th World Wide Web Conference*, pages 369–379. O'Reilly, 1995.