Trading-Off Type-Inference Memory Complexity Against Communication

Konstantin Hyppönen[1], David Naccache[2], Elena Trichina[1], and Alexei Tchoulkine[2]

[1] University of Kuopio
Department of Computer Science
Po.B. 1627, FIN-70211, Kuopio, Finland
{konstantin.hypponen, elena.trichina}@cs.uku.fi
[2] Gemplus Card International
Applied Research & Security Centre
34 rue Guynemer, Issy-les-Moulineaux, 92447, France
{david.naccache, alexei.tchoulkine}@gemplus.com

Abstract. While bringing considerable flexibility and extending the horizons of mobile computing, mobile code raises major security issues. Hence, mobile code, such as Java applets, needs to be analyzed before execution. The byte-code verifier checks low-level security properties that ensure that the downloaded code cannot bypass the virtual machine's security mechanisms. One of the statically ensured properties is *type safety*. The type-inference phase is the overwhelming resource-consuming part of the verification process.

This paper addresses the RAM bottleneck met while verifying mobile code in memory-constrained environments such as smart-cards. We propose to modify classic type-inference in a way that significantly reduces the memory consumption in the memory-constrained device at the detriment of its distrusted memory-rich environment.

The outline of our idea is the following, throughout execution, the memory frames used by the verifier are MAC-ed and exported to the terminal and then retrieved upon request. Hence a distrusted memory-rich terminal can be safely used for convincing the embedded device that the downloaded code is secure.

The proposed protocol was implemented on JCOP20 and JCOP30 Java cards using IBM's JCOP development tool.

1 Introduction

The Java Card architecture for smart cards [1] allows new applications, called *applets*, to be downloaded into smart cards. While general security issues raised by applet download are well known [9], transferring Java's safety model into resource-constrained devices such as smart cards appears to require the devising of delicate security-performance trade-offs.

When a Java class comes from a distrusted source, there are two basic manners to ensure that no harm will be done by running it.

The first is to interpret the code *defensively* [2]. A *defensive interpreter* is a virtual machine with built-in dynamic runtime verification capabilities. Defensive interpreters have the advantage of being able to run standard class files resulting from *any* Java compilation chain but appear to be slow: the security tests performed during interpretation slow-down each and every execution of the downloaded code. This renders defensive interpreters unattractive for smart cards where resources are severely constrained and where, in general, applets are downloaded rarely and run frequently.

Another method consists in running the newly downloaded code in a completely protected environment (*sandbox*), thereby ensuring that even hostile code will remain harmless. In this model, applets are not compiled to machine language, but rather to a virtual-machine assembly-language called *byte-code*.

Upon download, the applet's byte-code is subject to a static analysis called *byte-code verification* which purpose is to make sure that the applet's code is well-typed. This is necessary to ascertain that the code will not attempt to violate Java's security policy by performing ill-typed operations at runtime (*e.g.* forging object references from integers or calling directly API private methods). Today's *de facto* verification standard is Sun's algorithm [7] which has the advantage of being able to verify any class file resulting from any standard compilation chain. While the time and space complexities of Sun's algorithm suit personal computers, the memory complexity of this algorithm appears prohibitive for smart cards, where RAM is a significant cost-factor.

This limitation gave birth to a number of innovating workarounds:

Leroy [5, 6] devised a verification scheme which memory complexity equals the amount of RAM necessary to run the verified applet. Leroy's solution relies on off-card code transformations whose purpose is to facilitate on-card verification by eliminating the memory-consuming fix-point calculations of Sun's original algorithm.

Proof carrying code [11] (PCC) is a technique by which a side product of the full verification, namely, the final type information inferred at the end of the verification process (*fix-point*), is sent along with the byte-code to allow a straight-line verification of the applet. This extra information causes some transmission overhead, but the memory needed to verify a code becomes essentially equal to the RAM necessary to run it. A PCC off-card proof-generator is a rather complex software.

Various other *ad-hoc* memory-optimization techniques exist as well [10, 8].

Our results: The work reported in this paper describes an alternative byte-code verification solution. Denoting by $M_{\max}$ the number of variables claimed by the verified method and by J the number of jump targets in it, we show how to securely distribute the verification procedure between the card and the terminal so as to reduce the card's memory requirements from $\mathrm{O}(M_{\max}J)$ to $\mathrm{O}(J\log J + cM_{\max})$ where c is a small language-dependent constant or, when a higher communication burden is tolerable, to a theoretic $\mathrm{O}(\log J + cM_{\max})$.

The rest of the paper is organized as follows: the next section recalls Java's security model and Sun's verification algorithm with a specific focus on its *data-*

flow analysis part. The subsequent sections describe the new verification protocol, which implementation details are given in the last section.

2 Java Security

The *Java Virtual Machine (JVM) Specification* [7] defines the executable file structure, called the *class file* format, to which all Java programs are compiled. In a class file, the executable code of *methods* (Java methods are the equivalent of C functions) is found in *code-array* structures. The executable code and some method-specific runtime information (namely, the maximal operand stack size $S_{\max}$ and the number of local variables $L_{\max}$ claimed by the method[3]) constitute a *code-attribute*. We briefly overview the general stages that a Java code goes through upon download.

To begin with, the classes of a Java program are translated into independent class files at compile-time. Upon a load request, a class file is transferred over the network to its recipient where, at link-time, symbolic references are resolved. Finally, upon method invocation, the relevant method code is interpreted (run) by the JVM.

Java's security model is enforced by the *class loader* restricting what can be loaded, the *class file verifier* guaranteeing the safety of the loaded code and the *security manager* and *access controller* restricting library methods calls so as to comply with the security policy. Class loading and security management are essentially an association of lookup tables and digital signatures and hence do not pose particular implementation problems. Byte-code verification, on which we focus this paper, aims at predicting the runtime behavior of a method precisely enough to guarantee its safety without actually having to run it.

2.1 Byte-Code Verification

Byte-code verification [4] is a link-time phase where the method's run-time behavior is proved to be *semantically correct*.

The *byte-code* is the executable sequence of bytes of the code-array of a method's code-attribute. The byte-code verifier processes units of method-code stored as class file attributes. An initial byte-code verification pass breaks the byte sequence into successive instructions, recording the offset (*program point*) of each instruction. Some static constraints are checked to ensure that the byte-code sequence can be interpreted as a valid sequence of instructions taking the right number of arguments. As this ends normally, the receiver assumes that the analyzed file complies with the general syntactical description of the class file format.

Then, a second verification step ascertains that the code will only manipulate values which types are compatible with Java's safety rules. This is achieved by a type-based *data-flow analysis* which abstractly executes the method's byte-code,

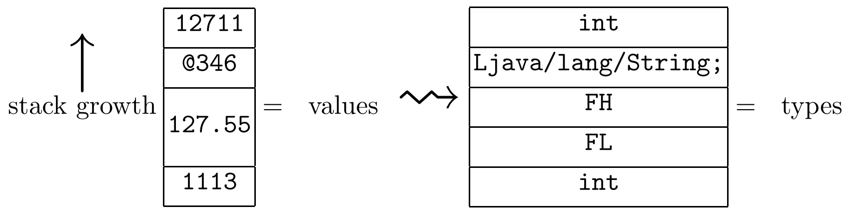
[3] $M_{\max} = L_{\max} + S_{\max}$.

by modelling the effect of the successive byte-codes on the *types* of the variables read or written by the code.

The next section explains the semantics of *type checking*, *i.e.*, the process of verifying that a given pre-constructed type is correct with respect to a given class file. We explain why and how such a type can always be constructed and describe the basic idea behind data-flow analysis.

The Semantics of Type Checking A natural way to analyze the behavior of a program is to study its effect on the machine's memory. At runtime, each program point can be looked upon as a memory *instruction frame* describing the set of all the runtime values possibly taken by the JVM's stack and local variables.

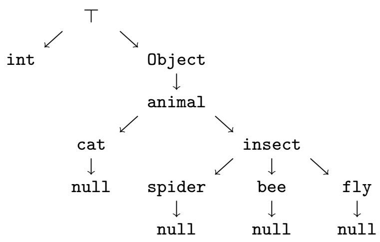
Since run-time information, such as actual input data is unknown before execution starts, the best an analysis may do is reason about *sets* of possible computations. An essential notion used for doing so is the *collecting semantics* defined in [3] where, instead of computing on a full *semantic* domain (values), one computes on a restricted *abstract* domain (types).



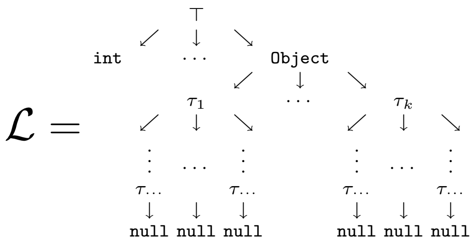
For reasoning with types, one must precisely classify the information expressed by types. A natural way to determine how (in)comparable types are is to rank all types in a *lattice* $\mathcal{L}$. A brief look at the toy lattice depicted below suffices to find-out that `animal` is more general than `fly`, that `int` and `spider` are not comparable and that `cat` is a specific `animal`. Hence, knowing that a variable is designed to safely contain an `animal`, one can infer that no harm can occur if during execution this variable would successively contain a `cat`, a `fly` and an `insect`. However, should the opposite be detected (*e.g.* an instruction would attempt to use a variable supposed to contain an `animal` as if it were a `cat`) the program should be rejected as unsafe.

The most general type is called *top* and denoted $\top$. $\top$ represents the *potential simultaneous presence of all types*, *i.e.* the *absence of (specific) information*. By definition, a special null-pointer type (denoted `null`) terminates the inheritance chain of all object descendants.

Formally, this defines a pointed complete partial order (CPO) $\preceq$ on the lattice $\mathcal{L}$.



Stack elements and local variable types are hence tuples of elements of $\mathcal{L}$ to which one can apply *point-wise ordering*.



Abstract Interpretation The verification process described in [7] §4.9, is an (iterative data-flow analysis) algorithm that attempts to build an *abstract description* of the JVM's memory for each program point. A byte-code is safe if the construction of such an abstract description succeeds.

Assume, for example, that an `iadd` is present at some program point. The `i` in `iadd` hints that this instruction operates on integers. `iadd`'s effect on the JVM is indeed very simple: the two topmost stack elements are popped, added and the sum is pushed back into the stack. An abstract interpreter will disregard the arithmetic meaning of `iadd` and reason with types: `iadd` pops two `int` elements from the stack and pushes back an `int`. From an abstract perspective, `iadd` and `isub` have identical effects on the JVM.

As an immediate corollary, a valid stack for executing an `iadd` *must* have a value which can be abstracted as $\texttt{int}.\texttt{int}.S$, where S may contain any sequence of types (which are irrelevant for the interpretation of our `iadd`). After executing `iadd` the stack becomes $\texttt{int}.S$

Denoting by L the JVM's local variable area (irrelevant to `iadd`), the total effect of `iadd`'s abstract interpretation on the JVM's memory can be described by the *transition rule* Φ:

$$\texttt{iadd}: \quad (\texttt{int}.\texttt{int}.S, \quad L) \quad \mapsto \quad (\texttt{int}.S, \quad L)$$

The following table defines the transition rules of seven representative JVM instructions[4].

[4] Note that the test $n \in L$ is equivalent to ascertaining that $0 \leq n \leq L_{\max}$.

Instruction	Transition rule Φ	Security test
iconst$[n]$	$(S,\quad L) \quad\mapsto (\texttt{int}.S,\quad L)$	$\mid S \mid < S_{\max}$
iload$[n]$	$(S,\quad L) \quad\mapsto (\texttt{int}.S,\quad L)$	$n \in L,\ L[n] == \texttt{int},\ \mid S \mid < S_{\max}$
istore$[n]$	$(\texttt{int}.S,\quad L) \quad\mapsto (S,\quad L\{n \to \texttt{int}\})$	$n \in L$
aload$[n]$	$(S,\quad L) \quad\mapsto (L[n].S,\quad L)$	$n \in L,\ L[n] \preceq \texttt{Object},\ \mid S \mid < S_{\max}$
astore$[n]$	$(\tau.S,\quad L) \quad\mapsto (S,\quad L\{n \to \tau\})$	$n \in L,\ \tau \preceq \texttt{Object}$
dup	$(\tau.S,\quad L) \quad\mapsto (\tau.\tau.S,\quad L)$	$\mid S \mid < S_{\max}$
getfield $C.f.\tau$	$(ref(D).S,\quad L) \mapsto (\tau.S,\quad L)$	$D \preceq C$

For the first instruction of the method, the local variables that represent parameters are initialized with the types τ_j indicated by the method's signature; the stack is empty (ϵ) and all other local variables are filled with $\top$s. Hence, the initial frame is set to:

$$(\epsilon, \quad (\texttt{this}, \tau_1, \ldots, \tau_{n-1}, \top, \ldots, \top))$$

For other instructions, no information regarding the stack or the local variables is available.

Verifying a method whose body is a straight-line code (no branches), is easy: we simply iterate the abstract interpreter' transition function Φ over the successive instructions, taking the stack and register types *after* any given instruction as the stack and register types *before* the next instruction. The types describing the successive JVM memory-states produced by the successive instructions are called *working frames*.

Denoting by $\mathrm{in}(i)$ the frame *before* instruction i and by $\mathrm{out}(i)$ the frame *after* instruction i, we get the following data-flow equation where evaluation starts from the right:

$$\mathrm{in}(i+1) \leftarrow \mathrm{out}(i) \leftarrow \Phi_i(\mathrm{in}(i))$$

Branches introduce forks and joins into the method's flowchart. Let us illustrate these with the following example:

program point	Java code
	`int m (int q) {`
$p_1 \hookrightarrow$	
	`    int x;`
	`    int y;`
	`    if (q == 0)`
$p_2 \hookrightarrow$	`         { x = 1; ... }`
$p_3 \hookrightarrow$	`    else { y = 2; ... }`
$p_4 \hookrightarrow$	
	`... }`

After program point p_1 one can infer that variable q has type `int`. This is denoted as $\mathrm{out}(p_1) = \{q = \texttt{int}, x = \top, y = \top\}$. After the `if`'s *then* branch, we infer the type of variable x, *i.e.*, $\mathrm{out}(p_2) = \{q = \texttt{int}, x = \texttt{int}, y = \top\}$. After the `else`, we learn that $\mathrm{out}(p_3) = \{q = \texttt{int}, x = \top, y = \texttt{int}\}$.

However, at p_4, nothing can be said about neither x nor y. We hence prudently assume that $\mathrm{in}(p_4) = \{q = \texttt{int}, x = \top, y = \top\}$ by virtue of the principle that if two execution paths yield different types for a given variable, only the lesser-information type can serve for further calculations. In other words, we assume the worst and check that, still, type-violations will not occur.

Thus, if an instruction i has several predecessors with different exit frames, i's frame is computed as the *least common ancestor*[5] (LCA) of all the predecessors' exit frames:

$$\text{in}(i) = \text{LCA}\{\text{out}(i) \mid j \in \text{Predecessor}(i)\}.$$

In our example: $\text{in}(p_4) = \{q = \texttt{int}, x = \top = \text{LCA}(\texttt{int}, \top), y = \top = \text{LCA}(\top, \texttt{int})\}$.

Finding an assignment of frames to program points which is sufficiently conservative for all execution paths requires testing them all; this is what the verification algorithm does. Whenever some $\text{in}(i)$ is adjusted, all frames $\text{in}(j)$ that depend on $\text{in}(i)$ have to be adjusted too, causing additional iterations until a *fix-point* is reached (*i.e.*, no more adjustments are required). The final set of frames is a *proof* that the verification terminated with success. In other words, that the byte-code is *well-typed*.

2.2 Sun's Type-Inference Algorithm

The algorithm below which summarizes the verification process, is taken from [7]. The treatment of exceptions (straightforward) is purposely omitted for the sake of clarity.

The *initialization* phase of the algorithm consists of the following steps:

1. Initialize $\text{in}(0) \leftarrow (\epsilon, \quad (\texttt{this}, \tau_1, \ldots, \tau_{n-1}, \top, \ldots, \top))$ where $(\tau_1, \ldots, \tau_{n-1})$ is the method's signature.
2. A 'changed' bit is associated to each instruction, all 'changed' bits are set to zero except the first.

Execute the following loop until no more instructions are marked as 'changed' (*i.e.*, a fix-point is reached).

1. Choose a marked instruction i. If there aren't any, the method is safe (exit). Otherwise, reset the 'changed' bit of the selected instruction.
2. Model the effect of the instruction on $\text{in}(i)$ by doing the following:
   - If the instruction uses values from the stack, ensure that:
     - There are sufficiently many values on the stack, and that
     - The topmost stack elements are of types that suit the executed instruction.

     Otherwise, verification fails.
   - If the instruction uses local variables:
     - Ascertain that these local variables are of types that suit the executed instruction.

     Otherwise, verification fails.

[5] The LCA operation is frequently called *unification*.

- If the instruction pushes values onto the stack:
  - Ascertain that there is enough room on the stack for the new values. If the new stack's height exceeds $S_{\max}$, verification fails;
  - Add the types produced by the instruction to the top of the stack.
- If the instruction modifies local variables, record these new types in $\text{out}(i)$.

3. Determine the instructions that can potentially follow instruction i. A successor instruction can be one of the following:
   - For most instructions, the successor instruction is just the next instruction;
   - For a `goto`, the successor instruction is the `goto`'s jump target;
   - For an `if`, both the `if`'s remote jump target and the next instruction are the successors;
   - `return` has no successors.
   - Verification fails if it is possible to "fall off" the last instruction of the method.
4. Unify $\text{out}(i)$ with the $\text{in}(k)$-frame of each successor instruction k.
   - If this successor instruction k is visited for the first time,
     - record that $\text{out}(i)$ calculated in step 2 is now the $\text{in}(k)$-frame of the successor instruction;
     - mark the successor instruction by setting the 'changed' bit.
   - If the successor instruction has been visited before,
     - Unify $\text{out}(i)$ with the successor instruction's (already present) $\text{in}(k)$-frame and update: $\text{in}(k) \leftarrow \text{LCA}(\text{in}(k), \text{out}(i))$.
     - If the unification caused modifications in $\text{in}(k)$, mark the successor instruction k by setting its 'changed' bit.
5. Go to step 1.

If the code is safe, the algorithm must exit without reporting a failure.

As one can see, the time complexity of this algorithm is upper-bound by the $\text{O}(D \times I \times J \times L_{\max})$, where D is the depth of the type lattice, I is the total number of instructions and J is the number of jumps in the method.

While from a theoretical standpoint, time complexity can be bounded by a crude upper bound $\text{O}(I^4)$[6], practical experiments show that each instruction is usually parsed less than twice during the verification process.

[6] In the worst case, all instructions are jumps, and each instruction acts on c different variables, *i.e.*, $L_{\max} = c \times I$, where c is a language-dependent constant representing the maximal number of variables possibly affected by a single instruction. Additionally, one may show (stemming from the observation that the definition of a new type requires at least one new instruction) that D is the maximal amongst the depth of the primitive data part of the type lattice $\mathcal{L}$ (some langauge-dependent constant) and I. This boils down to a crude upper bound $\text{O}(I^4)$. Considering that byte-code verification takes place only once upon applet downloading, even a relatively high computational overload would not be a barrier to running a byte-code verifier on board.

Space (memory) complexity is much more problematic, since a straightforward coding of Sun's algorithm yields an implementation where memory complexity is bound by $O(IL_{\max})$. Although this is still polynomial in the size of the downloaded applet, one must not forget that if $L_{\max}$ RAM cells are available on board for running applets, applets are likely to use up all the available memory so as to optimize their functional features, which in turn would make it impossible to verify these same applets on board. Here again, a straightforward simplification allows to reduce this memory complexity from $O(IL_{\max})$ to $O(JL_{\max})$.

3 Trading-Off On-Board RAM Against Communication

A smart card is nothing but one element in a distributed computing system which, invariably, comprises terminals (also called *card readers*) that allow cards to communicate with the outside world.

Given that terminals usually possess much more RAM than cards, it seems natural to rely on the terminal's storage capabilities for running the verification algorithm. The sole challenge being that data stored in the terminal's RAM can be subject to tampering.

Note that the capacity of working with remote objects (Remote Method Invocation) would make the implementation of such a concept rather natural in Java[7].

3.1 The Data Integrity Mechanism

Our goal being to use of the terminal's RAM to store the frames created during verification, the card must embark a mechanism allowing to ascertain that frame data is not modified without the card's consent. Luckily, a classic cryptographic primitive called MAC (Message Authentication Code) [12] does just that.

It is important to stress that most modern cards embark *ad hoc* cryptographic co-processors that allow the computation of MACs in a few clock cycles. The on-board operation of such co-processors is particularly easy through the cryptographic classes and Java Card's standard APIs. Finally, the solution that we are about to describe does not impose upon the terminal any cryptographic computations; and there is no need for the card and the terminal to share secret keys.

Before verification starts, the card generates an ephemeral MAC key k; this key will be used only for one method verification. We denote by $f_k(m)$ the MAC function, applied to data m. k should be long enough (typically 160 bits long) to avoid the illicit recycling of data coming from different runs of the verification algorithm.

The protocol below describes the solution implemented by our prototype. In the coming paragraphs we use the term *working frame*, when speaking of

[7] However, because of the current limitations of Java Cards, the prototype reported in this paper does not rely on RMIs.

$\text{in}(i+1) \leftarrow \text{out}(i) \leftarrow \Phi_i(\text{in}(i))$. In other words, the working frame is the current input frame $\text{in}(i+1)$ of the instruction i which is just about to be modelled.

For simplicity, we assume that instruction number i is located at offset i. Shouldn't this be the case, a simple lookup table $A[i]$, which output represents the real offset of the i-th instruction, will fix the problem.

The card does not keep the frames of the method's instructions in its own RAM but uses the terminal as a repository for storing them. To ascertain data integrity, the card sends out, along with the data, MACs of the outgoing data. These MACs will subsequently allow the card to ascertain the integrity of the data retrieved from the terminal (in other words, the card simply sends MACs *to itself* via the terminal).

The card associates with each instruction i a counter c_i kept in card's RAM. Each time that instruction i is rechecked (modelled) during the fix-point computation, its c_i is incremented inside the card. The role of c_i is to avoid playback attacks, *i.e.* the malicious substitution of type information by an older versions of this type information.

3.2 The New Byte-code Verification Strategy

The *initialize* step is replaced by repeating the following for $2 \leq i \leq I$:

1. Form a string representing the initialized (void) type information (frame) F_i for instruction i.
2. Append to this string a counter c_i representing the current number of times that instruction i was visited. Start with $c_i \leftarrow 0$.
3. Compute $r_i = f_k(\mathsf{unchanged}, c_i, i, F_i) = f_k(\mathsf{unchanged}, 0, i, F_i)$.
4. Send to the terminal $\{\mathsf{unchanged}, F_i, i, r_i\}$.

Complete the initialization step by:

1. Sending to the terminal $\{\mathsf{changed}, F_1 \leftarrow (\epsilon, \quad (\mathtt{this}, \tau_1, \ldots, \tau_{n-1}, \top, \ldots, \top)), 1, r_1 \leftarrow f_k(\mathsf{changed}, c_1 \leftarrow 0, 1, F_1)\}$,
2. Initializing an on-board counter $\tau \leftarrow 1$.

In all subsequent descriptions *check* r_i means: re-compute r_i based on the current i, the $\{c_i, k\}$ kept in the card and $\{F_i, \mathsf{changed}/\mathsf{unchanged}\ \text{bit}\}$ sent back by the terminal and if the result disagrees with the r_i sent back by the terminal, reject the applet.

The main fix-point loop is the following:

1. If $\tau = 0$ accept the applet, else query from the terminal an F_i for an instruction i which bit is set to $\mathsf{changed}$.
   (a) Check if the transition rules allow executing the instruction. In case of failure reject the applet.
   (b) Apply the transition rules to the type information F_i received back from the terminal and store the result in the working frame.

2. For all potential successors j of the instruction at i:
   (a) Query the terminal for $\{F_j, r_j\}$; check that r_j is correct.
   (b) Unify the working frame with F_j. If unification fails reject the applet.
   (c) If unification yields a frame F'_j different than F_j then
      – increment c_j, increment τ
      – compute $r_j = f_k(\mathsf{changed}, c_j, j, F'_j)$, and
      – send to the terminal $\{\mathsf{changed}, F'_j, j, r_j\}$.

   The terminal can now erase the old values at entry j and replace them by the new ones.
3. Decrement τ, increment c_i, re-compute r_i and send $\{\mathsf{unchanged}, F_i, i, r_i\}$ to the terminal. Again, the terminal can now erase the old values at entry i and replace them by the new ones.
4. Goto 1.

The algorithm that we have just described only requires the storage of I c_i-counters. Since time complexity will never exceed $\mathrm{O}(I^4)$, any given instruction can never be visited more than $\mathrm{O}(I^4)$ times. The counter size can hence be bound by $\mathrm{O}(\log I)$ thereby resulting in an overall on-board space complexity of $\mathrm{O}(I \log I + cL_{\max})$. where c is a small language-dependent constant (the $cL_{\max}$ component of the formula simply represents the memory space necessary for the working frame).

Note that although in our presentation we allotted for clarity a c_i per instruction, this is not actually necessary since the same c_i can be shared by every sequence of instructions into which no jumps are possible; this $\mathrm{O}(J \log J + cL_{\max})$ memory complexity optimization is evident to Java verification practitioners.

3.3 Reducing In-card Memory to $\mathbf{O}(\log I + cL_{\max})$

By exporting also the c_i values to the terminal, we can further reduce card's memory requirements to $O(\log I + cL_{\max})$. This is done by implementing the next protocol in which all the c_i values are kept in the terminal.

The card generates a second ephemeral MAC key k' and stores a single counter t, initialized to zero.

– **Initialization:** The card computes and sends $m_i \leftarrow f_{k'}(i, c_i \leftarrow 0, t \leftarrow 0)$ to the terminal for $1 \leq i \leq I$.
– **Read** c_i**:** To read a counter c_i:
  - The card sends a query i to the terminal.
  - The terminal returns $\{c_i, m_i\}$.
  - The card checks that $m_i = f_{k'}(i, c_i, t)$ and if this is indeed the case then c_i can be used safely (in case of MAC disagreement the card rejects the applet).
– **Increment** c_i**:** to increment a counter c_i:
  1. For $j = 1$ to I:
     - Execute **Read** c_j

- If $i = j$, the card instructs the terminal to increment c_i.
- The card computes $m_j = f_{k'}(j, c_j, t+1)$ and sends this updated m_j to the terminal.

2. The card increments t.

The value of t being at most equal to the number of steps executed by the program, t occupies an $\mathrm{O}(\log I)$ space (in practice, a 32 bit counter). Note, however, that the amount of communication and computations is rather important: for every c_i update, the terminal has to send back to the card the values and MACs of *all* counters associated with the verified method; the card checks all the MACs, updates them correspondingly, and sends them back to the terminal.

4 Implementation Details

We implemented algorithm 3.2 as a usual Java Card applet. It is uploaded onto the card and after initialization, waits a new applet to be received in order to check it for type safety. Thus, our prototype does not have any access to the Java Card Runtime Environment (JCRE) structures nor to Installer's functions and by no means can it access information about the current contents of the card and packages residing on it. However, the purpose of our code is to check the type safety of newly uploaded applets. Given that new applets can make use of packages already existing on board, our verifier should have full information about the following structures:

– the names of the packages already present on board and classes in these packages;
– methods for resident classes, along with their signatures;
– fields in resident classes and their types.

Since this information cannot be obtained from the card itself, we had to assume that the newly downloaded applet uses only common framework packages, and pre-embed the necessary information about these packages into our verifier.

The type lattice information is “derived” by the verifier from the superclass references and interface references stored in the byte arrays of classes.

The terminal-side applet plays an active role in the verification process; it calls methods of the card-side applet and sends them all the necessary data.

4.1 Programming Tools and Libraries

The prototype has been implemented as a “normal” Java Card applet. It enjoys the full functionality of Sun's off-card verifier, that we reverse-engineered in the course of this project using a special application called `dump`, from the `JTrek` library [13] originally developed by Compaq[8].

`JTrek` contains the `Trek` class library, which allows navigation and manipulation of Java class files, as well as several applications built around this library;

[8] JTrek is no longer downloadable from its web page.

`dump` being one such application. `dump` creates a text file containing requested information for each class file of the *trek* (*i.e.*, a path through a list of class files and their objects); in particular, the generated text file may contain class file's attributes, instructions, constant pool, and source statements. All this makes it possible to reconstruct source code from class files.

After decompiling the program class file (and fixing some of `JTrek`'s bugs in a process) we obtained, amongst other things:

- Parsers for the Java Card CAP and export files;
- The verifier's static checks for all JCVM byte codes;
- An abstract interpreter for the methods including the representation of the JCVM states.

These tools were used to develop the terminal-side verifier applet; and some ideas were recycled for developing the card-side verifier applet.

For actual applet development we used IBM Zurich Research Laboratory's `JCOP Tools` [14]. This toolbox consists of the `JCOP IDE` (Integrated Development Environment) and `BugZ`, a source-level debugger. Furthermore, shell-like APDU command execution environment, as well as command-driven `CardMan` are included for simple card management tasks, such as listing packages and applets installed on the card, displaying information about given CAP files, installing applets from an uploaded package, sending arbitrary APDU commands to the card, etc.

`JCOP Tools` are shipped with the off-card Application Programming Interface (API). Using the provided implementations of these APIs, it is possible to develop applications that can:

- Upload the CAP file onto a card;
- Install the applet on a card;
- Communicate with the card's applet (*i.e.*, send APDUs to the applet and receive APDUs from it);
- Delete the applet instance and the package from the card.

Since `JCOP Tools` can interact with any Java Card inserted into the reader, the availability of cryptographic functions depends on the card. The kit is shipped with three Java Cards; all of which support 3DES encryption/decryption, and two support RSA.

Hence, the `JCOP Tools` provided us with all the necessary features for implementing both the card-side and the terminal-side parts of our protocol, testing them on virtual as well as real Java Cards and allowing to benchmark the whole.

4.2 Interaction Between Terminal-Side and Card-Side Applets

The implemented prototype consists of the terminal-side and card-side applets. Both applets run in parallel.

The verification algorithm is fully deterministic (with the exception of the selection of a single frame from the set of all frames marked as `changed`). Since the order in which marked frames are selected does not affect the final result (*i.e.*, accept or reject the applet), the terminal-side applet can be "proactive" because it has all necessary information for running the verification process in parallel with the card[9]. Using this strategy, we can avoid all requests from the card to the terminal given that the latter is fully aware of the current verification state and can hence provide the card-side applet with all required data without being prompted.

Thus, the only data sent from the card to the terminal are response status and MAC-ed frames that have to be stored in the terminal. The terminal initiates all verification steps; it sends the card the results of the modelling of each instruction and the results of unification of different frames. The card-side applet simply checks that the verification process advances as it should and updates the instruction counters[10].

The Terminal-Side Applet The terminal-side applet is based on Sun Microsystems' off-card verifier. The latter was fully revised and some new functionality added. The communication with the card-side applet is implemented using IBM JCOP's API.

The terminal-side applet is in charge of the following tasks:

– Prepare the CAP file components for sending them to the card-side applet. Parse the CAP file (storing it in the object structure) and check its compliance with Sun's file format (structural verification being beyond the scope of our demonstrator, we left this part off-board for the time being);
– Maintain the storage for frames and their MACs. Exchange frames with the card-side applet;
– Resolve the problem of finding the LCA of two frames in nontrivial cases (trivial ones can be dealt with by our card-side applet) and send the result to the card.

The Card-Side Applet The card-side applet:

– Controls the correctness of the verifier's method calls by the terminal-side applet;
– Checks and applies transition rules (*i.e.*, performs type inference) to individual instructions.
– Maintains a list of counters c_i for all instructions; updates counter values as necessary;
– Executes cryptographic functions;

[9] Note that this *is not* along the general design philosophy of our protocol whereby the terminal needs no other form of intelligence other than the capacity to receive data, store it and fetch it back upon request. We nonetheless implemented some extra intelligence in the terminal to speed-up the development of our proof of concept.

[10] Again, the previous footnote applies to this simplification as well.

– Solves the problem *Is type A a descendant of type B in the type lattice $\mathcal{L}$?* (in other words, is $A \preceq B$?) in order to check the result of the unification of two frames sent by the terminal;
– For instructions `invokespecial`, `invokestatic` and `invokevirtual`, checks arguments for their type consistency and pushes the returned type onto the operand stack. Supports calls to all framework methods as well as to methods of the package being currently verified. The `invokeinterface` instruction is not yet supported.
– The card-side applet can unify two frames for all types of stack and local variables except when both types to be unified are references to classes or arrays of references to classes. In this case, the card-side applet asks the terminal to perform unification, waits for results, and checks these results before accepting.

5 Conclusion

Our proof-of-concept (not optimized) implementation required 380 Kbytes for the terminal-side applet source code and 70 Kbyte for the card-side applet source code. With the maximum length of method's byte-code set to 200 bytes and both, $S_{\max}$ and $L_{\max}$ limited to 20 (the restrictions of the Java Cards shipped with `JCOP Tools`), one needs 440 bytes of RAM to run our two-party verification procedure. When the verified byte-code is written into EEPROM (as is the case in most real-life scenarios), one would need only 240 bytes of on-board RAM and 8976+ 200 EEPROM bytes.

The natural way to turn our prototype into a full-fledged verifier, is to incorporate it into the *Installer* applet, which has already its own representation of the CAP file components.

We do not think that communication overhead is a serious concern. With the advent of fast card interfaces, such as USB, the transmission's relative cost is reduced. Typically, USB tokens can feature various performances ranging from a 1.5 Mb/s (low-speed) to 12 Mb/s (full speed). But even with slower interfaces, such as ISO 7816-3 our prototype still functions correctly in real-time.

References

1. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.
2. R. Cohen, *The defensive Java virtual machine specification*, Technical Report, Computational Logic Inc., 1997.
3. P. Cousot, R. Cousot, *Abstract Interpretation: a Unified Lattice Model for Static Analysis by Construction or Approximation of Fixpoints*, Proceedings of POPL'77, ACM Press, Los Angeles, California, pp. 238-252.
4. X. Leroy, *Java Byte-Code Verification: an Overview*, In G. Berry, H. Comon, and A. Finkel, editors, Computer Aided Verification, CAV 2001, volume 2102 of Lecture Notes in Computer Science, pp. 265-285, Springer-Verlag, 2001.

5. X. Leroy, *On-Card Byte-code Verification for Java card*, In I. Attali and T. Jensen, editors, Smart Card Programming and Security, proceedings E-Smart 2001, volume 2140 of Lecture Notes in Computer Science, pp. 150-164, Springer-Verlag, 2001.
6. X. Leroy, *Byte-code Verification for Java smart card*, Software Practice & Experience, 32:319-340, 2002.
7. T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley, 1999.
8. N. Maltesson, D. Naccache, E. Trichina, C. Tymen *Applet Verification Strategies for RAM-constrained Devices*, In Pil Joong Lee and Chae Hoon Lim, editors, Information Security and Cryptology – ICISC 2002, volume 2587 of Lecture Notes in Computer Science, pp. 118-137, Springer-Verlag, 2002.
9. G. McGraw, E. Felten *Java Security*, John Wiley & Sons, 1999.
10. D. Naccache, A. Tchoulkine, C. Tymen, E. Trichina *Reducing the Memory Complexity of Type–Inference Algorithms*, In R. Deng, S. Qing, F. Bao and J. Zhou, editors, Information and Communication Security, ICICS 2002, volume 2513 of Lecture Notes in Computer Science, pp. 109-121, Springer-Verlag, 2002.
11. G. Necula, *Proof-carrying code*, Proceedings of POPL'97, pp. 106-119, ACM Press, 1997.
12. B. Schneier, *Applied Cryptography: Second Edition: protocols, algorithms and source code in C*, John Willey & Sons, 1996.
13. `http://www.digital.com/java/download/jtrek/`
14. `http://www.zurich.ibm.com/jcop/news/news.html`