Applet Verification Strategies for RAM-Constrained Devices

[Published in P.J. Lee and C.H. Lim, Eds., *Information Security and Cryptology – ICISC 2002*, vol. 2587 of *Lecture Notes in Computer Science*, pp. 118–137, Springer-Verlag, 2003.]

Nils Maltesson[1], David Naccache[2], Elena Trichina[1], and Christophe Tymen[2]

[1] Lund Institute of Technology
Magistratsvägen 27A, Lund, 226 43, Sweden
d99nm@efd.lth.se, nmaltesson@hotmail.com
[2] Gemplus Card International
34 rue Guynemer, Issy-les-Moulineaux, 92447, France
{david.naccache, christophe.tymen}@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 memory consumption.

Our algorithm is inspired by bit-slice data processing and consists in running the verifier on each variable in turn. In other words, instead of running the fix-point calculation algorithm once on M variables, we re-launch the algorithm M/ℓ times, verifying each time only ℓ variables. Parameter ℓ can then be tuned to suit the RAM resources available on board whereas M/ℓ upper-bounds the computational effort (expressed in re-runs of the usual fix-point calculation algorithm). The resulting RAM economy, as experimented on a number of popular applets, is around 40%.

1 Introduction

The Java Card architecture for smart cards [2] allows new applications, called *applets*, to be downloaded into smart-cards. While bringing considerable flexibility and extending the horizons of smart-card usage this *post issuance* feature raises major security issues. Upon their loading, malicious applets can try to

subvert the JVM's security in a variety of ways. For example, they might try to overflow the stack, hoping to modify memory locations which they are not allowed to access, cast objects inappropriately to corrupt arbitrary memory areas or even modify other programs (Trojan horse attacks). While the 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* [3]. 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; as will be seen later, the memory complexity of these tests is not negligible either. This renders defensive interpreters unattractive for smart-cards where resources are severely constrained and were, 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. Java's security model is based on sandboxes. The sandbox is a neutralization layer preventing direct access to hardware resources. 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 [8] 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 [6, 7] 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.

The work reported in this paper describes two new memory-optimization techniques.

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 in detail our algorithms, which benchmarks are given in the last section.

2 Java Security

The *Java Virtual Machine (JVM) Specification* [8] 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) 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 [5] 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.

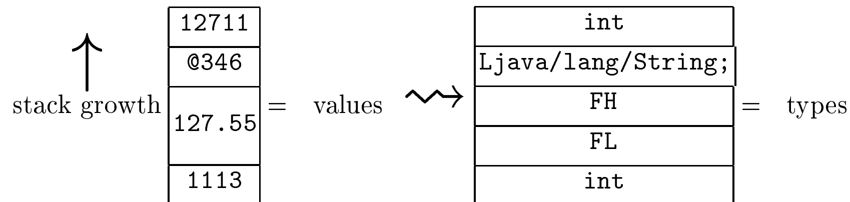
As this ends normally, the receiver assumes that the analyzed file complies with the general syntactical description of the class file format.

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, by modeling 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.

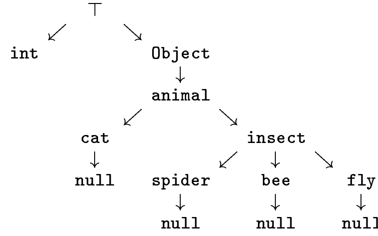
Since run-time information, such as actual input data is unknown before execution starts, the best an analysis may do is reason about *sets* of possible computations. An essential notion used for doing so is the *collecting semantics* defined in [4] where, instead of computing on a full *semantic* domain (values), one computes on a restricted *abstract* domain (types).



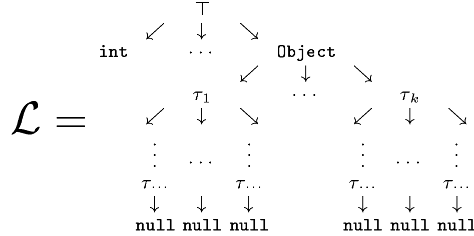
For reasoning with types, one must precisely classify the information expressed by types. A natural way to determine how (in)comparable types are is to rank all types in a *lattice* $\mathcal{L}$. A brief look at the toy lattice depicted below suffices to find-out that `animal` is more general than `fly`, 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 [8] 4.9, is an (iterative data-flow analysis) algorithm that attempts to builds 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.[3]

[3] 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`$[n]$	$(S, \quad L)$	$\mapsto (\texttt{int}.S, \quad L)$	$\mid S \mid < S_{\max}$
`iload`$[n]$	$(S, \quad L)$	$\mapsto (\texttt{int}.S, \quad L)$	$n \in L$, $L[n] == \texttt{int}$, $\mid S \mid < S_{\max}$
`istore`$[n]$	$(\texttt{int}.S, \quad L)$	$\mapsto (S, \quad L\{n \rightarrow \texttt{int}\})$	$n \in L$
`aload`$[n]$	$(S, \quad L)$	$\mapsto (L[n].S, \quad L)$	$n \in L$, $L[n] \preceq \texttt{Object}$, $\mid S \mid < S_{\max}$
`astore`$[n]$	$(\tau.S, \quad L)$	$\mapsto (S, \quad L\{n \rightarrow \tau\})$	$n \in L$, $\tau \preceq \texttt{Object}$
`dup`	$(\tau.S, \quad L)$	$\mapsto (\tau.\tau.S, \quad L)$	$\mid S \mid < S_{\max}$
`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 *frames*.

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

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

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 $\text{out}(p_1) = \{q = \texttt{int}, x = \top, y = \top\}$. After the `if`'s *then* branch, we infer the type of variable x, *i.e.*, $\text{out}(p_2) = \{q = \texttt{int}, x = \texttt{int}, y = \top\}$. After the `else`, we learn that $\text{out}(p_3) = \{q = \texttt{int}, x = \top, y = \texttt{int}\}$.

However, at p_4, nothing can be said about neither x nor y. We hence prudently assume that $\text{in}(p_4) = \{q = \texttt{int}, x = \top, y = \top\}$ 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*[4] (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 [8]. 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:

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

- Ascertain that these local variables are of types that suit the executed instruction.

  Otherwise, verification fails.
- 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.

2.3 Basic blocks and memory complexity

As explained above, the data-flow type analysis of a straight-line code consists of simply applying the transition function to the sequence of instructions $i_1, i_2, ..., i_t$ taking $\text{in}(i_k) \leftarrow \text{out}(i_{k-1})$. This property can be used for optimizing the algorithm.

Following [1, 10], we call a *basic block* ($\mathbb{B}$) a straight-line sequence of instructions that can be entered only at its beginning and exited only at its end. For instance, we identify in the example below four basic blocks denoted $\mathbb{B}_0, \mathbb{B}_1, \mathbb{B}_2$ and $\mathbb{B}_3$:

```
Public class Example {
   public int cmpz (int a, int b)
                                  {
       int c;
       if (a==b)
          c = a+b;
       else
          c = a*a;
       return c;
   }
}
```

compile →

```
     Method int cmpz(int,int)
B0  0      iload_1
B0  1      iload_2
B0  2      if_cmpne 12
B1  5      iload_1
B1  6      iload_2
B1  7      iadd
B1  8      istore_3
B1  9      goto 16
B2 12      iload_1
B2 13      iload_1
B2 14      imul
B2 15      istore_3
B3 16      iload_3
B3 17      ireturn
```

In several implementations of Sun's algorithm, the data-flow equations evolve at the basic-block-level rather than at the instruction-level. In other words, it suffices to keep track in permanent memory only the frames $\text{in}(i)$ where i is the first instruction of a $\mathbb{B}$ (*i.e.*, a branch target). All other frames within a basic block can be temporarily recomputed on the fly. By extension, we denote by $\text{in}(\mathbb{B})$ and $\text{out}(\mathbb{B})$, the frames before and after the execution of $\mathbb{B}$. The entire program is denoted by $\mathbb{P}$.

Denoting by N_{blocks} the number of $\mathbb{B}$s in a method, a straightforward implementation of Sun's algorithm allocates N_{blocks} frames, each of size $L_{\max} + S_{\max}$.

$L_{\max}$ and $S_{\max}$ are determined by the compiler and appear in the method's header. This results in an $\mathcal{O}((L_{\max} + S_{\max}) \times N_{\text{blocks}})$ memory-complexity. In practice, the verification of moderately complex methods would frequently require a few thousands of bytes.

2.4 The stack's behavior

A property of Java code is that a *unique* stack height is associated to each program point. This property is actually verified on the fly during type-inference although it could be perfectly checked *independently* of type-inference.

In other words, the computation of stack heights does not require the modeling of the instructions' effect on types, but only on the *stack-pointer*.

Denoting by σ_i the stack height associated to program point i, this section presents a simple algorithm for computing $\{\sigma_0, \sigma_1, \ldots\}$ from $\mathbb{P}$

The algorithm uses a table Δ associating to each instruction a signed integer indicating the effect of this instruction on the stack's size:

Δ	Instruction	Δ	Instruction	Δ	Instruction	Δ	Instruction
2	iconst[n]	1	sconst[n]	1	bspush	2	bipush
1	aload	1	sload	1	aload[n]	2	iload[n]
-1	aaload	0	iaload	-1	astore	-2	istore
-1	astrore[n]	-2	store[n]	-1	pop	1	dup
-1	sadd,smul	-2	iadd,imul	0	getfield_a	1	getfield_i
0	iinc	-3	icmp	-1	ifne	-2	if_acmpne
0	goto	0	return	0	athrow	0	arraylength

The information we are looking for is easily obtained by running Sun's algorithm with the modeling effect on types *turned off*, monitoring only the code's effect on the stack pointer:

Algorithm PredictStack($\mathbb{P}$)

- Associate to each program point i a bit changed[i] indicating if this program point needs to be re-examined; initialize all the changed[i]-bits to zero.
- Set $\sigma_0 \leftarrow 0$; changed[0] $\leftarrow 1$;
- For all exception code entry points[5] j, set changed[j] $\leftarrow 1$; $\sigma_j \leftarrow 1$;
- While $\exists\ i$ such that changed[i] $== 1$:
  - Set changed[i] $\leftarrow 0$;
  - $\alpha \leftarrow \sigma_i + \Delta(i)$
  - If $\alpha > S_{\max}$ or $\alpha < 0$ then report a failure.
  - If i is the program's last instruction and it is possible to fall-off the program's code then report a failure.
  - For each successor instruction k of i:
    - If k is visited for the first time then set $\sigma_k \leftarrow \alpha$; changed[$k$] $\leftarrow 1$
    - If k was visited before and $\sigma_k \neq \alpha$, then report a failure.
- Return $\{\sigma_0, \sigma_1, \ldots\}$

3 A Simplified Defensive Virtual Machine Model

We model the JVM by a very basic state-machine. Although over-simplified, our model suffices for presenting the verification strategies described in this paper.

3.1 Memory elements

Variables and the stack elements will be denoted:

$$L = \{L[0], \ldots, L[L_{\max} - 1]\} \quad \text{and} \quad S = \{S[0], \ldots, S[S_{\max} - 1]\}$$

Since in Java a precise stack height σ_j is associated with each j we can safely use a unique memory-space M to identify all memory elements: albeit, the stack

[5] These can be found in method_component.exception_handlers[j].handler_offset fields of Java card *.cap files.

machine can be very easily converted into a full register machine by computing $\{\sigma_0, \sigma_1, \ldots\} \leftarrow \mathsf{PredictStack}(\mathbb{P})$ and replacing stack accesses $S[\sigma_j]$ by register accesses $L[L_{\max} + \sigma_j]$.

we thus denote $M_{\max} = L_{\max} + S_{\max}$ and:

$$M = \{M[0], \ldots, M[M_{\max} - 1]\} = \{L[0], \ldots, L[L_{\max} - 1], S[0], \ldots, S[S_{\max} - 1]\}$$

3.2 Operational semantics

We assume that each instruction reads and re-writes the entire memory M. In other words, although in reality only the contents of very few variables will change after the execution of each byte-code, we regard the byte-code at program point j as a collection of $M_{\max}$ functions:

$$M[i] \leftarrow \phi_{j,i}(M) \quad \text{for} \quad 0 \leq i < M_{\max}$$

which collective effect can be modeled as:

$$M \leftarrow \{\phi_{j,0}(M), \ldots, \phi_{j,M_{\max}-1}(M)\} = \Phi_j(M)$$

Based upon the instruction (j) and the data (M) the machine selects a new j (the current instruction's successor) using an additional "next instruction" function $\theta_j(M)$.

Execution halts when $\theta_j(M)$ outputs a special value denoted `stop`.

Using the above notation, the method's execution boils-down to setting $j \leftarrow 0$ and iterating $\{j, M\} \leftarrow \{\theta_j(M), \Phi_j(M)\}$ while $j \notin \{\mathtt{stop}, \mathtt{error}_{\mathtt{runtime}}\}$.

where $\mathtt{error}_{\mathtt{runtime}}$ signals an error encountered during the course of execution (such as a division by zero for instance).

3.3 Defensive interpretation

A Defensive JVM associates to each value $M[i]$ a type denoted $\bar{M}[i] \in \mathcal{L}$. In general, functions and variables operating on types will be distinguished by upper bars ($\bar{V}$ represents the type of the value contained in V).

Given an instruction j, Java's tying rules express the effect of j on $\bar{M}$ through a function:

$$\bar{\Phi}_j(\bar{M}) : \mathcal{L}^{M_{\max}} \mapsto \{\mathcal{L} \cup \mathtt{error}_{\mathtt{type}}\}^{M_{\max}}$$

where $\mathtt{error}_{\mathtt{type}}$ is an error resulting from a violation of Java's typing rules. By definition, whenever $\mathtt{error}_{\mathtt{type}}$ occurs, execution stops.

The effect of $\bar{\Phi}_j$ simply shadows that of Φ_j:

$$\bar{M} \leftarrow \{\bar{\phi}_{j,0}(\bar{M}), \ldots, \bar{\phi}_{j,M_{\max}-1}(\bar{M})\} = \bar{\Phi}_j(\bar{M})$$

The complete Defensive Java Virtual Machine $\mathsf{DJVM}(\mathbb{P}, \text{input data})$, can hence be modeled as follows:

- $\{j, M, \bar{M}\} \leftarrow \{0, \text{input data}, \text{signature}(\mathbb{P})\}$
- while ($j \notin \{\texttt{stop}, \texttt{error}_{\texttt{runtime}}\}$ and $\texttt{error}_{\texttt{type}} \notin \bar{M}$)
  - $\{j, M, \bar{M}\} \leftarrow \{\theta_j(M), \Phi_j(M), \bar{\Phi}_j(\bar{M})\}$

4 Variable-Wise Verification

Variable-wise verification is inspired by bit-slice data processing and consists in running the verifier on each variable *in turn*. In other words, instead of calculating at once the fix-points of $M_{\max}$ variables, we launch the algorithm $M_{\max}/\ell$ times, verifying each time only ℓ variables. Parameter ℓ can then be tuned to suit the RAM resources available on board whereas $M_{\max}/\ell$ will upper-bound the computational effort expressed in re-runs of [8].

The advantage of this approach is the possibility to re-use the *same* tiny RAM space for the sequential verification of *different* variables.

4.1 A toy-example

Consider the following example where $\ell = 1$, and the operation

$$M[13] \leftarrow M[4] + M[7] \tag{1}$$

is to be verified.

The operator $+$ (`sadd`) requires two arguments of type `short`; we launch the complete verification process for $i \leftarrow 0, \ldots, M_{\max} - 1$:

- When $i \notin \{4, 7, 13\}$ nothing is done.
- When $i = 4$ (*i.e.* we are verifying $M[4]$), the algorithm meets expression (1) and *only* checks that $\bar{M}[4]$ is `short`, assuming that $\bar{M}[7]$ is `short`. The operator's effect on $M[13]$ is ignored.
- When i reaches 7, the algorithm meets expression (1) again and checks *only* that $\bar{M}[7]$ is `short`, this time the algorithm assumes that $\bar{M}[4]$ is `short`. The operator's effect on $M[13]$ is ignored again.
- When i reaches 13, the algorithm meets expression (1) and models its effect *only* on $\bar{M}[13]$ by assigning $\bar{M}[13] \leftarrow$ `short`.

Hence, in runs 4 and 7 we successively ascertained that no type violations occurred in the first ($M[4]$, run 4) or the second ($M[7]$, run 7) argument of the operator $+$, while the 13-th round modeled the effect of `sadd` on $M[13]$.

Note that the same RAM variable could be used to host, in turn, the type information associated to $M[4]$, $M[7]$ and $M[13]$.

4.2 The required properties

For this to work, each instruction (j) must comply with the following two properties:

1. There exist $M_{\max} - 1$ sets of types $\mathcal{T}_{j,0}, \ldots, \mathcal{T}_{j,M_{\max}-1}$ such that:

$$\forall \bar{M} \in \mathcal{T}_{j,0} \times \mathcal{T}_{j,1} \times \ldots \times \mathcal{T}_{j,M_{\max}-1}, \quad \texttt{error}_{\texttt{type}} \notin \bar{\Phi}_j(\bar{M})$$

2. $\forall \bar{M}, \bar{M}' \in \mathcal{T}_{j,0} \times \mathcal{T}_{j,1} \times \ldots \times \mathcal{T}_{j,M_{\max}-1}$

$$\forall i, 0 \leq i < M_{\max}, \quad \bar{M}[i] = \bar{M}'[i] \Rightarrow \bar{\phi}_{j,i}(\bar{M}) = \bar{\phi}_{j,i}(\bar{M}')$$

The first requirement expresses the *independence* between the types of variables *read* by the instruction; this is necessary to verify independently each variable regardless the types of its neighbors. The second requirement (*self-sufficiency*) guarantees that the type of each variable before executing the instruction suffices to precisely determine its type after the execution of the instruction.

4.3 Byte-code compliance

We now turn to examine the compliance of a few concrete Java-card [2] byte-codes with these definitions. The stack elements that our examples will operate on are:

$$\begin{aligned}&\{S[\sigma_j], S[\sigma_j+1], S[\sigma_j+2], \ldots\} = \\ &\quad \{M[L_{\max}+\sigma_j], M[L_{\max}+\sigma_j+1], M[L_{\max}+\sigma_j+2], \ldots\}\end{aligned}$$

Example 1: `icmp icmp` transforms the types of the four topmost stack elements from {`intH`, `intL`, `intH`, `intL`} to {`short`, `undef`, `undef`, `undef`}.

(1) is fulfilled: the sets from which variable types can be chosen are:

$$\text{for} \quad i \notin \{0,1,2,3\} \quad \mathcal{T}_{j,L_{\max}+\sigma_j+i} = \mathcal{L}$$

$$\begin{aligned}\mathcal{T}_{j,L_{\max}+\sigma_j} &= \{\texttt{intH}\} \qquad & \mathcal{T}_{j,L_{\max}+\sigma_j+1} &= \{\texttt{intL}\} \\ \mathcal{T}_{j,L_{\max}+\sigma_j+2} &= \{\texttt{intH}\} \qquad & \mathcal{T}_{j,L_{\max}+\sigma_j+3} &= \{\texttt{intL}\}\end{aligned}$$

(2) is also fulfilled: the type of each variable after the execution of `icmp` can be precisely determined from the variable's type before executing `icmp`:

$$\text{for} \quad i \notin \{0,1,2,3\} \quad \bar{\phi}_{j,L_{\max}+\sigma_j+i}(\bar{M}) = \bar{M}[L_{\max}+\sigma_j+i]$$

$$\begin{aligned}\bar{\phi}_{j,L_{\max}+\sigma_j}(\bar{M}) &= \texttt{short} \qquad & \bar{\phi}_{j,L_{\max}+\sigma_j+1}(\bar{M}) &= \texttt{undef} \\ \bar{\phi}_{j,L_{\max}+\sigma_j+2}(\bar{M}) &= \texttt{undef} \qquad & \bar{\phi}_{j,L_{\max}+\sigma_j+3}(\bar{M}) &= \texttt{undef}\end{aligned}$$

Example 2: `pop` `pop` acts only on the topmost stack element (namely, $S[\sigma_j] = M[L_{\max} + \sigma_j]$) and transforms its type from any type different than `intL` to `undef`.

property (1): $$\mathcal{T}_{j,x} = \begin{cases} \mathcal{L} - \{\texttt{intL}\} & \text{for} \quad x = L_{\max} + \sigma_j \\ \mathcal{L} & \text{for} \quad x \neq L_{\max} + \sigma_j \end{cases}$$

property (2): $$\bar{\phi}_{j,L_{\max}+\sigma_j+i}(\bar{M}) = \begin{cases} \texttt{undef} & \text{for} \quad i = 0 \\ \bar{M}[L_{\max} + \sigma_j + i] & \text{for} \quad i \neq 0 \end{cases}$$

Example 3: `dup` `dup` duplicates the topmost stack element $S[\sigma_j] = M[L_{\max} + \sigma_j]$. Property (1) is satisfied (`dup` can duplicate any type):

$$\mathcal{T}_{j,0} \times \mathcal{T}_{j,1} \times \ldots \times \mathcal{T}_{j,M_{\max}-1} = \mathcal{L}^{M_{\max}}$$

However, property (2) is clearly violated for $L_{\max} + \sigma_j + 1$; indeed, an M and an M' such that $\bar{M}[L_{\max} + \sigma_j] \neq \bar{M}'[L_{\max} + \sigma_j]$ and $\bar{M}[L_{\max} + \sigma_j + 1] = \bar{M}'[L_{\max} + \sigma_j + 1] = \texttt{undef}$, yield:

$$\bar{\phi}_{L_{\max}+\sigma_j+1}(\bar{M}) = \bar{M}[L_{\max} + \sigma_j] \neq \bar{\phi}_{L_{\max}+\sigma_j+1}(\bar{M}') = \bar{M}'[L_{\max} + \sigma_j]$$

Hence, unlike the previous examples, `dup` does not lend itself to variable-wise verification. `dup` belongs to a small family of byte-codes (namely: `dup`, `dup2`, `dup_x`, `swap_x`, `aload`, `astore` and `athrow`) that 'mix' or 'cross-contaminate' the types of the variables they operate on.

The workaround is simple: before starting verification, parse $\mathbb{P}$. Whenever one of these problematic instructions is encountered, group all the variables processed by the instruction into one, bigger, 'extended' variable. The algorithm performing this packing operation, $\mathsf{Group}(\mathbb{P})$, is described in the next section.

4.4 Grouping variables

Grouping transforms the list $\mathfrak{M} = \{0, 1, 2, \cdots, M_{\max} - 1\}$ into a list $\mathfrak{G}$ with a lesser number of symbols. All $\mathfrak{G}$-elements containing equal symbols are to be interpreted as $M[i]$ cells that must be verified *together* as their types are *interdependent*.

The algorithm below describes the grouping process. Although in our practical implementation $\mathsf{PredictStack}(\mathbb{P})$ was merged into $\mathsf{Group}(\mathbb{P})$'s main loop (this spares the need to save $\sigma_0, \sigma_1, \ldots$), $\mathsf{PredictStack}(\mathbb{P})$ was moved here into the initialization phase for the sake of clarity.

Algorithm Group($\mathbb{P}$)

- Initialize $\mathfrak{M} \leftarrow \{0, 1, 2, \cdots, M_{\max} - 1\}$. For the sake of simplicity, we denote by $\mathfrak{S}[i]$ the elements of $\mathfrak{M}$ that shadow stack cells and by $\mathfrak{L}[i]$ the elements of $\mathfrak{M}$ that shadow local variables[6].
- An 'unseen' bit is associated to each instruction. All 'unseen' bits are reset.
- Run PredictStack($\mathbb{P}$) to compute $\sigma_0, \sigma_1, \ldots$

Iterate the following until no more 'unseen' bits are equal to zero (*i.e.*, all the method's byte-codes were processed exactly once):

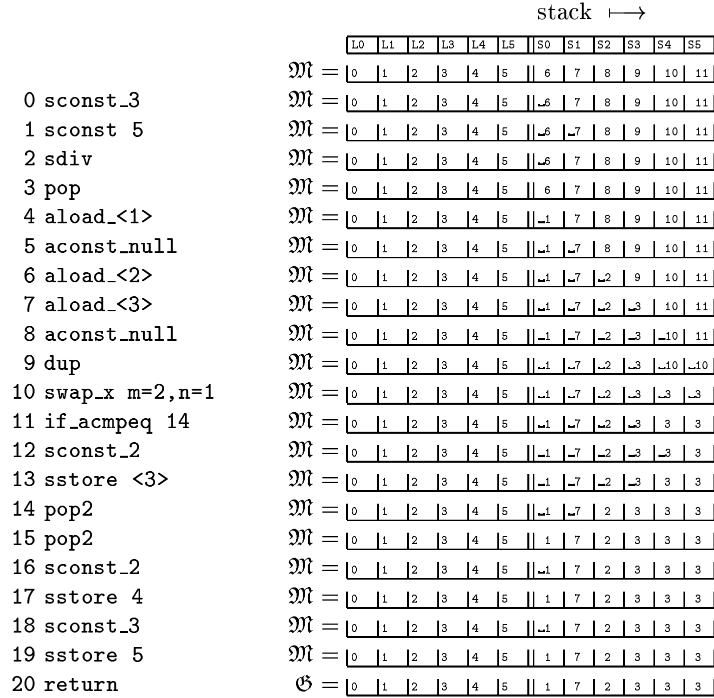
- Choose an 'unseen' instruction j. If there aren't any return the list $\mathfrak{G} \leftarrow \mathfrak{M}$ and exit. Otherwise, set the 'unseen' bit of the selected instruction.
  - if the j-th instruction is a `dup`, `dup2`, `dup_x` or `swap_x` then lookup the row $\ell(k)$ corresponding to instruction j. For all non-empty entries in $\ell(k)$ replace all occurrences of $\max\{\mathfrak{S}[\sigma_j + \ell(k)], \mathfrak{S}[\sigma_j + k]\}$ in $\mathfrak{M}$ by $\min\{\mathfrak{S}[\sigma_j + \ell(k)], \mathfrak{S}[\sigma_j + k]\}$.

bytecode $\downarrow$	$k \rightarrowtail$	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7
`dup_x` $\{m=1, n=1,0\}$	$\ell(k) \rightarrowtail$				0								
`dup_x` $\{m=1, n=2\}$	$\ell(k) \rightarrowtail$				0		0						
`dup_x` $\{m=1, n=3\}$	$\ell(k) \rightarrowtail$				0		0	0					
`dup_x` $\{m=1, n=4\}$	$\ell(k) \rightarrowtail$				0		0	0	0				
`dup_x` $\{m=1, n=5\}$	$\ell(k) \rightarrowtail$				0		0	0	0	0			
`dup_x` $\{m=2, n=5\}$	$\ell(k) \rightarrowtail$			0	0		0	0	0	0			
`dup_x` $\{m=3, n=5\}$	$\ell(k) \rightarrowtail$		0	0	0		0	0	0	0			
`dup_x` $\{m=3, n=4\}$	$\ell(k) \rightarrowtail$		0	0	0		0	0	0				
`dup_x` $\{m=2, n=3\}$	$\ell(k) \rightarrowtail$			0	0		0	0					
`dup_x` $\{m=4, n=7\}$	$\ell(k) \rightarrowtail$	0	0	0	0		0	0	0	0	0	0	
`dup_x` $\{m=4, n=5\}$	$\ell(k) \rightarrowtail$	0	0	0	0		0	0	0	0			
`dup_x` $\{m=3, n=7\}$	$\ell(k) \rightarrowtail$		0	0	0		0	0	0	0	0	0	
`dup_x` $\{m=2, n=2,0\}$	$\ell(k) \rightarrowtail$			0			1						
`dup_x` $\{m=2, n=4\}$	$\ell(k) \rightarrowtail$			0			1	0	1				
`dup_x` $\{m=2, n=6\}$	$\ell(k) \rightarrowtail$			0			1	0	1	0	1		
`dup_x` $\{m=4, n=6\}$	$\ell(k) \rightarrowtail$	0	1	0			1	0	1	0	1		
`dup_x` $\{m=3, n=3,0\}$	$\ell(k) \rightarrowtail$		0				2	1					
`dup_x` $\{m=3, n=6\}$	$\ell(k) \rightarrowtail$		0				2	1	0	2	1		
`dup_x` $\{m=4, n=8\}$	$\ell(k) \rightarrowtail$	0					3	2	1	0	3	2	1
`dup_x` $\{m=4, n=4,0\}$	$\ell(k) \rightarrowtail$	0					3	2	1				
`dup`	$\ell(k) \rightarrowtail$				0								
`dup2`	$\ell(k) \rightarrowtail$			0			1						
`swap_x` $\{m=1, n=1\}$	$\ell(k) \rightarrowtail$						0						
`swap_x` $\{m=1, n=2\}$	$\ell(k) \rightarrowtail$						0	0					
`swap_x` $\{m=2, n=1\}$	$\ell(k) \rightarrowtail$						0	0					
`swap_x` $\{m=2, n=2\}$	$\ell(k) \rightarrowtail$							0	-1				

  - if the j-th instruction is an `aload_<n>`, `astore_<n>`, `aload <n>`, or `astore <n>` then replace all occurrences of $\max\{\mathfrak{L}[n], \mathfrak{S}[\sigma_j]\}$ in $\mathfrak{M}$ by $\min\{\mathfrak{L}[n], \mathfrak{S}[\sigma_j]\}$.
  - if the j-th instruction is an `athrow` then replace all occurrences of $\max\{\mathfrak{S}[0], \mathfrak{S}[\sigma_j]\}$ in $\mathfrak{M}$ by $\min\{\mathfrak{S}[0], \mathfrak{S}[\sigma_j]\}$.

The process is illustrated below by a toy-example where the character '␣' denotes stack cells used by the program.

[6] *i.e.* $\mathfrak{L}[i] \stackrel{\text{def}}{=} \mathfrak{M}[i]$ and $\mathfrak{S}[i] \stackrel{\text{def}}{=} \mathfrak{M}[i + L_{\max}]$.



Given that the largest group of variables (those tagged by 3) has four elements (namely L3, S3, S4 and S5), it appears that the code can be verified with 4-cell frames (instead of 12-cell ones).

Having reduced memory complexity as much as we could, it remains to determine how many passes are required to verify the code. At a first glance, seven passes will do, namely:

pass 1	L3 S3 S4 S5
pass 2	L2 S2
pass 3	L1 S0
pass 4	L0
pass 5	L4
pass 6	L5
pass 7	S1

However, given that we anyway pay the price of a 4-cell memory complexity, it would be a pity to re-launch the entire verification process without packing passes 2, 3, 4, 5, 6 and 7 into two additional 4-cell passes. For instance:

pass 1	L3 S3 S4 S5
pass 2	L2 S2 L4 L5
pass 3	L1 S0 L0 S1

This is realized by the algorithm described in the next section.

4.5 Bin-packing

Bin-packing is the following NP-complete problem:

Given a set of n positive integers $\mathfrak{U} = \{u_1, u_2, \ldots, u_n\}$ and a positive bound B, divide $\mathfrak{U}$ into k disjoint subsets $\mathfrak{U} = \mathfrak{U}_1 \cup \mathfrak{U}_2 \cup, \ldots, \cup \mathfrak{U}_k$ such that:

- The sum of all elements in each subset $\mathfrak{U}_i$, denoted $\sum \mathfrak{U}_i$ is smaller than B.
- The number of subsets k is minimal.

Although no efficient algorithm can solve this problem exactly, a number of efficient algorithms that find very good approximate solutions (*i.e.* $k' \approxeq k$) exist [15–17].

Bin-packing (approximation) algorithms come in two flavors: on-line and off-line ones. On-line algorithms receive the u_is one after another and place u_i in a subset *before* getting u_{i+1}. Although the on-line constraint is irrelevant to our case (we dispose of the entire set $\mathfrak{U}$ as Group($\mathbb{P}$) ends), very simple on-line algorithms [14] computing approximations tighter than $k \leq k' \leq \frac{17}{10}k + 2$ exist.

First-Fit: places u_i in the leftmost $\mathfrak{U}_j$ that has enough space to accommodate u_i. If no such $\mathfrak{U}_j$ is found, then a new $\mathfrak{U}_j$ is opened.

Best-Fit: places u_i in the $\mathfrak{U}_j$ that u_i fills-up the best. In other words, u_i is added to the $\mathfrak{U}_j$ that minimizes $B - \sum \mathfrak{U}_j - u_i$. In case of tie, the lowest index j is chosen. If no such $\mathfrak{U}_j$ is found, then a new $\mathfrak{U}_j$ is opened.

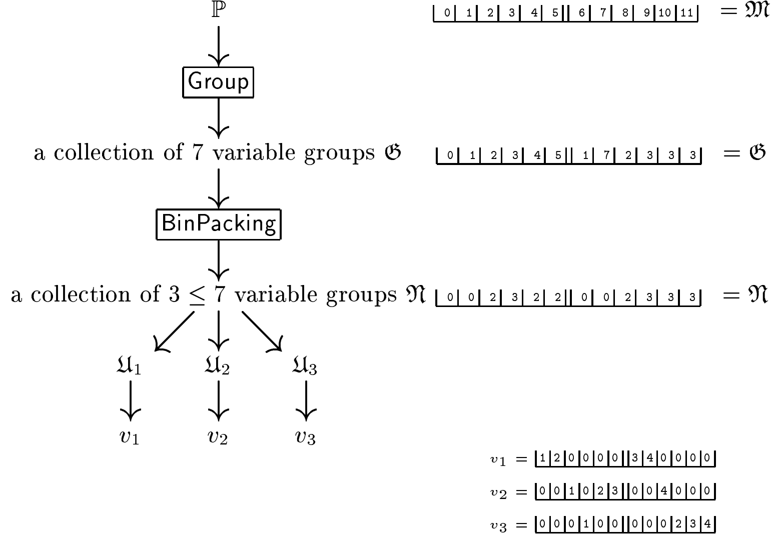
Refined versions of these algorithms (*e.g.* Yao's First-Fit) even find approximations tighter than $k \leq k' \leq \frac{5}{3}k + 5$.

Off-line algorithms perform much better. Best-fit and First-Fit can be improved by operating on a *sorted* $\mathfrak{U}$. In other words, the biggest u_i is placed first, then the second-biggest u_i is placed *etc.* The resulting algorithms are called First-Fit-Decreasing and Best-Fit-Decreasing and yield approximations tighter than $k \leq k' \leq \frac{11}{9}k + 4$.

Note that the implementation of both Best-Fit-Decreasing and First-Fit-Decreasing on 8-bit micro-controllers are trivial. We denote by $\{v_1, \ldots, v_k\} \leftarrow$ BinPacking($\mathfrak{G}$) the following algorithm:

- Let u_i be the number of occurrences of symbol i in $\mathfrak{G}$. Let $B = \max\{u_i\}$. Initialize $\mathfrak{N} \leftarrow \mathfrak{G}$.
- Solve $\{\mathfrak{U}_1, \ldots, \mathfrak{U}_k\} \leftarrow$ BestFitDecreasing($B; \{u_1, \ldots, u_n\}$)
- For $i \leftarrow 1$ to k:
  if u_j was placed in $\mathfrak{U}_i$ then replace all occurrences of j in $\mathfrak{N}$ by $\beta_i = \min_{u_j \in \mathfrak{U}_i} \{j\}$.
- Let $\{v_1, v_2, \cdots, v_k\}$ be a set of $M_{\max}$-bit strings initialized to zero.
- For $i \leftarrow 1$ to k
  - $w \leftarrow 1$
  - For $\ell \leftarrow 0$ to $M_{\max} - 1$
    if $\mathfrak{N}[\ell] == \beta_i$ set $v_i[\ell] \leftarrow w$; set $w \leftarrow w + 1$;
- Return $\{v_1, v_2, \cdots, v_k\}$

Hence, the effect of BinPacking on the previous example's output would be:



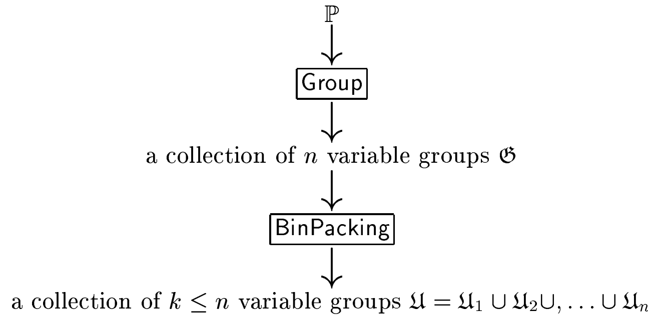
4.6 Putting the pieces together

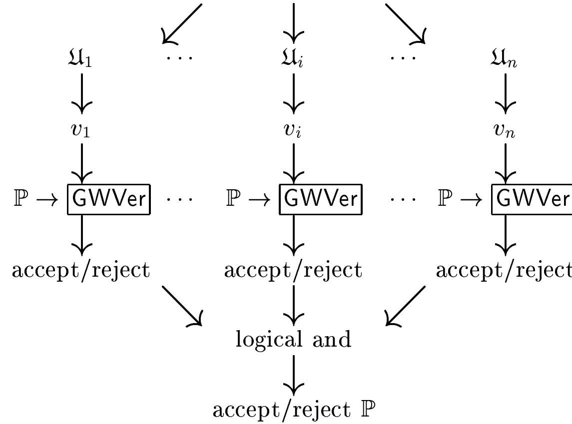
The group-wise verification process $\mathsf{GWVer}(\mathbb{P}, v)$ mimics very closely Sun's original algorithm. There are only two fundamental differences between the two algorithms:

- In $\mathsf{GWVer}(\mathbb{P}, v)$ each frame contains only $\mu = \max(v[i])$ memory cells (denoted $\bar{\text{in}}(i) = \{T[0], \ldots, T[\mu - 1]\}$) instead of M_{max}-cell frames.
- Whenever Sun's verifier reads or writes a variable $M[i]$ in some $\text{in}(\cdot)$, then $\mathsf{GWVer}(\mathbb{P}, v)$ substitutes this operation by a reading or a writing into the memory cell $T[v[i] - 1]$ in $\bar{\text{in}}(i)$.

Hence, we built a memory interface to Sun's algorithm so that execution would require $\mathcal{O}(\mu \times N_{\text{blocks}})$ memory-complexity instead of a $\mathcal{O}(M_{\text{max}} \times N_{\text{blocks}})$

The entire process is summarized in the following schematic:





To evaluate experimentally the above process, we wrote a simple program that splits variables into categories for a given `*.jca` file and counts the number of RAM cells necessary to verify its most greedy method. We used for our estimates the representative Java card applets from [13]. The detailed outputs of our program are available upon request from the authors. Results are rather encouraging, the new verification strategy seems to roughly save 40% of the memory claimed by [8]. Increase in workload is a rough doubling of verification time (due to more complex bookkeeping and the few inherent extra passes traded-off against memory consumption).

Applet	**Sun [8]**	**Group-Wise**
`NullApp.jca`	6	$4 = 6 \times 66\%$
`HelloWorld.jca`	40	$12 = 40 \times 30\%$
`JavaLoyalty.jca`	48	$45 = 48 \times 93\%$
`Wallet.jca`	99	$55 = 99 \times 55\%$
`JavaPurse.jca`	480	$200 = 480 \times 41\%$
`Purse.jca`	550	$350 = 550 \times 63\%$
`CryptoApplet.jca`	4237	$2230 = 4237 \times 52\%$

Acknowledgments

The authors would like to thank Jacques Stern for his help concerning a number of technical points.

References

1. A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.

3. R. Cohen, *The defensive Java virtual machine specification*, Technical Report, Computational Logic Inc., 1997.
4. 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.
5. 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.
6. 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.
7. X. Leroy, *Bytecode Verification for Java smart card*, Software Practice & Experience, 32:319-340, 2002.
8. T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley, 1999.
9. G. McGraw, E. Felten *Securiy Java*, John Wiley & Sons, 1999.
10. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
11. G. Necula, *Proof-carrying code*, Proceedings of POPL'97, pp. 106-119, ACM Press, 1997.
12. D. Schmidt, *Denotational Semantics, a Methodology for Language Development*, Allyn and Bacon, Boston, 1986.
13. P. Bieber, J. Cazin, A. El-Marouani, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon, *The PACAP prototype: a tool for detecting java card illegal flows*, In I. Attali and T. Jensen, editors, Java on Smart Cards: Programming and Security, vol. 2041 of Lecture Notes in Computer Science, pp. 25-37, Springer-Verlag, 2001.
14. A. Yao, *New algorithms for bin packing*, Journal of the ACM, 27(2):207-227, April 1980.
15. W. de la Vega, G. Lueker, *Bin packing can be solved within $1+\epsilon$ in linear time*, Combinatorica, 1(4):349-355, 1981.
16. D. Johnson, A. Demers, J. Ullman, M. Garey, R. Graham, Worst-case performance bounds for simple one-dimensional packaging algorithms, SIAM Journal on Computing, 3(4):299-325, December 1974.
17. B. Baker, *A new proof for the first-fit decreasing bin-packing algorithm*, SIAM Journal Alg. Disc. Meth., 2(2):147-152, June 1981.