Computing stack maps with interfaces

Frédéric Besson[1], Thomas Jensen[2], and Tiphaine Turpin[3]

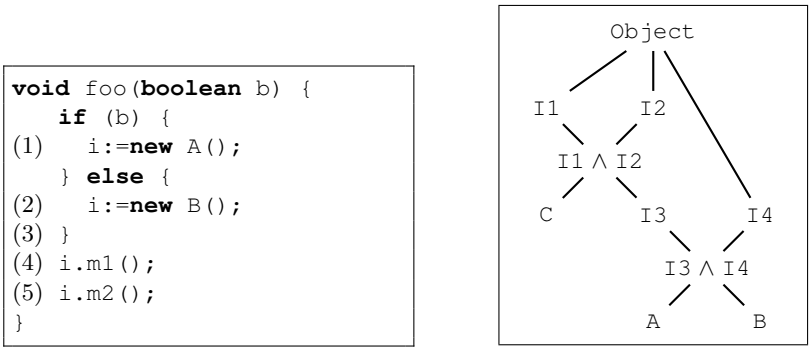
[1] Inria
[2] CNRS
[3] Université de Rennes I
first.last@irisa.fr

Abstract. Lightweight bytecode verification uses stack maps to annotate Java bytecode programs with type information in order to reduce the verification to type checking. This paper describes an improved bytecode analyser together with algorithms for optimizing the stack maps generated. The analyser is simplified in its treatment of base values (keeping only the necessary information to ensure memory safety) and enriched in its representation of interface types, using the Dedekind-MacNeille completion technique. The computed interface information allows to remove the dynamic checks at interface method invocations. We prove the memory safety property guaranteed by the bytecode verifier using an operational semantics whose distinguishing feature is the use of untagged 32-bit values. For bytecode typable without sets of types we show how to prune the fix-point to obtain a stack map that can be checked without computing with sets of interfaces *i.e.*, lightweight verification is not made more complex or costly. Experiments on three substantial test suites show that stack maps can be computed and correctly pruned by an optimized (but incomplete) pruning algorithm.

1 Introduction

The Java bytecode verifier, which is part of the Java Virtual Machine (JVM) [13], is a central component of Java security. At load time, the verifier checks that the bytecode conforms to the JVM typing policy. Together with additional dynamic checks this enables the virtual machine to run safely untrusted bytecodes such as web applets or mobile phone midlets. While the standard bytecode verifier performs a dataflow analysis on the bytecode the lightweight bytecode verifier [4] only checks the analysis result (which is called a *stack map*) that is shipped with the bytecode. It was originally designed for resource-constrained devices but the mainstream Java 2 Standard Edition (J2SE) is now moving towards lightweight bytecode verification, with slightly enhanced stack maps (see JSR 202 [9]).

A particular issue for the type inference performed in bytecode verification is the possibility for a class to implement several interfaces. The problem arises as soon as the language has multiple inheritance (only for interfaces, in the case of Java). This implies that the type hierarchy is not a lattice and prevents the computation of a unique most precise type for some variables, unless using sets of types. For simplicity, the choice made in the original verifiers (both standard



(a) Main method (b) Type hierarchy

Fig. 1: A Java bytecode program and its type hierarchy

and lightweight) was to ignore interfaces in bytecode verification and to make the necessary checks dynamically. This choice has been maintained in JSR 202.

We propose to extend the bytecode analysis to check interfaces statically, using conjunctions of types, and then to prune the result to get a stak map without conjunctions that can be fed to an almost unmodified checker. This does not work for every bytecode, but it applies to bytecode obtained by compilation of Java programs. As a result, the dynamic checks on interface methods may be safely removed for free. We describe the case of (idealised) Java bytecode, but the solution would apply to a more general use of multiple inheritance.

In this paper, the term analysis refers to the typing process that produces stack maps, checking is the validation of those stack maps on the consumer's side, and verification encompasses analysing, possibly pruning, and checking.

Motivating example Figure 1 provides a small example which illustrates the existing verification and its extension to conjunctive types. Figure 1a represents a bytecode program written in pseudo Java, without type information. We suppose a type hierarchy with three classes `A`, `B` and `C` and four interfaces `I`i ($i \in [1, \ldots, 4]$) where `C` implements and `I3` extends `I1` and `I2`, and `A` and `B` implements `I3` and `I4`. Each interface `I`i declares a method `m`i. Figure 1b shows the completion of the type hierarchy that is used by our enhanced analyser, which adds the elements `I1` $\wedge$ `I2` and `I3` $\wedge$ `I4` to the type hierarchy.

The standard bytecode verifier ignores interfaces. Thus, in method `foo`, the variable `i` at program point 3 is given as type the first common super-class of `A` and `B`, *i.e.*, `Object`. Note also that the call to each method `m`i is in fact a call to the method of interface `I`i, where it is declared. When analysing those calls, the bytecode verifier only checks statically that the variable `i` contains an object type. At run-time, the JVM dynamically checks whether the object referenced by `i` implements `I1` and `I2`, before doing a lookup with respect to the dynamic type of `i`. If it is not the case, a run-time exception is thrown.

Our extended analizer will type the example program using conjunctions of types, and in particular, the variable `i` at program point 3 will have type `I3` $\wedge$ `I4`, which is propagated at program points 4 and 5. As this is a sub-type of both `I1` and `I2`, this ensures that the two method calls are safe. However, for the purpose of lightweight bytecode verification, it is desirable to avoid annotating the variables with conjunctions. The backtracking pruning algorithm proposed in Section 4 detects that in the above conjunction, only `I3` is needed to type the subsequent method invocations, hence it removes `I4`. In the resulting stack map, the variable `i` has therefore the type `I3` at program points 3, 4 and 5.

The example also shows that opting for a backward program analysis does not simplify the problem. An analyser which starts from the invocation sites and propagates these "uses" of a variable to the point of definition would still require the use of conjunctions and lead to a back-tracking algorithm. With such a technique, the variable `i` at program point 5 would get the type `I2`. The problem arises when typing `i` at program point 4: it must be the intersection of `I1` and `I2`, which requires either to introduce the conjunction `I1` $\wedge$ `I2`, or to choose one of the types `C` and `I3`. The right choice can only be made knowing the creation sites 1 and 2, hence the need for a backtraking algorithm.

Organisation of the paper The formal development of these ideas is done in Sections 2–4. We define the intraprocedural part of a small-step operational semantics with big-step calls for a subset of the Java bytecode (Section 2), with a low-level treatment of values that allows for a convincing definition of the memory safety property (which is the base for all other security properties). The analysis is presented in Section 3 in terms of an abstract interpretation [5]. We define the notions of stack maps and lightweight verification for a method, and state the main soundness theorem of the stack maps produced by the analysis. Section 4 describes the pruning algorithms that remove conjunctive types from those stack maps. We give an efficient algorithm that works in all but some well-identified, pathological cases, that do not seem to occur in average Java programs. We report on some experiments on verifying Soot, Eclipse and a suite of Java MIDP applets for mobile phones with a prototype implementation in Section 5. Related work is discussed in Section 6 and Section 7 concludes.

Notations Sets have long *italic* names, other constants and constants functions are in roman, with the exception of bytecode instructions for which we use sans serif. Meta-variables have short lowercase *italic* names, except that we write $C, I, F^\sharp$ for classes, interfaces and abstract transfer functions, respectively.

For sets a and b, we write $a \rightharpoonup b$ for the set of partial functions from a to b. If f is a partial function, $\mathrm{dom}(f)$ is the domain of f, and any boolean expression e containing a sub-expression $f(e')$ implicitely means: $e' \in \mathrm{dom}(f) \ \wedge e$. We note $|x|$ for the cardinal of the set x, or the cardinal of its domain if x is a function. If f is a function (or a partial function), we note $f[x \leftarrow v]$ the function that maps x to v and any $y \neq x$ to $f(y)$.

Cartesian product takes precedence over other set operations: $\times \ \prec \ \cup, \rightarrow, \rightharpoonup$.

2 Intraprocedural semantics and memory safety

In this section we define formally our bytecode language and its semantics, and state the memory safety property that we ensure. We define the semantics (and safety) of one single method, parameterised by the semantics of the (direct) method calls it may involve. The interprocedural part of the semantics (which is essentially defined as a least fix-point of the intraprocedural semantics) and the proof that the safety property can be lifted to whole programs are omitted for space reasons. Details can be found in section 4 of a companion technical report [2].

2.1 The Java bytecode language

We present a minimal subset of the language, abstracting away irrelevant features (such as the operand stack) while keeping the main aspects (objects, interface methods) that are relevant to typing. The subset is sufficiently representative for the results to extend to the whole Java bytecode. We list some features that are absent from our language. The local operand stack is a feature of the bytecode that has no impact on the abstract domain used for the verification. For this reason, we replace actual bytecode instructions by three-address style instructions that directly operate on local variables. Constant fields and static or interface members, exceptions, void methods, basic types others than int, sub-routines, threads, class and objects initialisation, access control (visibility), as well as explicit type checks (the checkcast instruction) are not considered. We discuss their impact on this study in Section 5.1. Also, for conciseness we use less specialised instructions than the real JVM (for example, we merge iaload and aaload, ireturn and areturn).

Let *ident* be the set of fully qualified Java identifiers. We assume a set $class \subset ident$ of class names with a distinguished element Object, and a disjoint set $interface \subset ident$ of interface names. We define the set of types recursively as:

$$type ::= \text{int} \quad | \quad C \quad | \quad I \quad | \quad t[]$$

where $C \in class$, $I \in interface$, and $t \in type$.

The class hierarchy is modeled by the following three functions:

$$\begin{array}{lrl} \text{super} & : class \setminus \{\text{Object}\} & \rightarrow \quad class \\ \text{implements} & : \quad class & \rightarrow \mathcal{P}(interface) \\ \text{extends} & : \quad interface & \rightarrow \mathcal{P}(interface) \end{array}$$

The function super must be the ancestor function of a tree with root Object:

$$\forall C \in class \quad \exists i \geq 0 \quad \text{super}^i(C) = \text{Object}.$$

A method signature is made of an object type, an identifier and a list of parameter types. We assume a subset *msig* of such signatures which represents the methods that are declared in the program being verified:

$$msig \subset \{t.m(t_1, \ldots, t_n) \mid t \in type \setminus \{\text{int}\}, m \in ident, t_i \in type\}$$

Note that we have a virtual method signature if t is a class or an array type, and an interface method signature otherwise. We let $\text{arity}(t.m(t_1, \ldots, t_n)) = n$. Each method signature has a return type given by the function $\text{result} : \mathit{msig} \to \mathit{type}$.

A field signature is made of a class name, an identifier and a type. We assume a subset fsig of such signatures:

$$\mathit{fsig} \subset \{C.f : t \mid C \in \mathit{class}, f \in \mathit{ident}, t \in \mathit{type}\}$$

Note that $C.f : t$ represents a field declared in class C, and consequently, the set of fields that are relevant for a given class of objects must be looked for in its super-classes too (see the function fields in Section 2.2).

As stated in the introduction, we only consider the verification of one "current" method. We write var for the set of local variables of the method to verify and $\mathit{arg} \subseteq \mathit{var}$ for its set of formal parameters, whose types are described by the function $\mathrm{t}_{\mathrm{arg}} : \mathit{arg} \to \mathit{type}$. The return type of the current method is denoted by $\mathrm{t}_{\mathrm{ret}} \in \mathit{type}$. Program points are represented by the interval $\mathit{ppoint} = [0, |\mathit{ppoint}| - 1]$. Expressions and instructions are defined as follows. Here, C ranges over classes, t over types, ms over method signatures, fs over field signatures, x, y, z, x_i over local variables, and p over program points.

$$\begin{array}{rcl}
\mathit{expr} & ::= & n \qquad n \in [-2^{31}, 2^{31} - 1] \\
& \mid & \mathsf{null} \;\mid\; y + z \;\mid\; \mathsf{new}\ C \;\mid\; y.fs \;\mid\; \mathsf{new}\ t[y] \;\mid\; y[z] \\
& \mid & y.ms(x_1, \ldots, x_{\mathrm{arity}(ms)})
\end{array}$$

$$\begin{array}{rcl}
\mathit{instr} & ::= & x := e \qquad e \in \mathit{expr} \\
& \mid & x.fs := y \;\mid\; x[y] := z \;\mid\; \mathsf{goto}\ p \;\mid\; \mathsf{if}\ x < y\ \ p \;\mid\; \mathsf{return}\ x
\end{array}$$

The method code is represented by the function $\text{code} : \mathit{ppoint} \to \mathit{instr}$ mapping program points to instructions. The last instruction $\text{code}(|\mathit{ppoint}| - 1)$ must be either $\mathsf{goto}\ p$ for some p or $\mathsf{return}\ x$ for some x.

These last definitions enforce some well-formedness constraints on the code: the execution remains within the bounds of the code and cannot fall through the end, only valid local variables are referred to, and methods are always called with the right number of arguments. These properties are normally checked by the bytecode verifier prior to the type verifications.

2.2 Semantics

The operational semantics is defined as a small-step transition relation between program states (except for method calls which are big-step). A distinguishing feature of our semantic model is that we use a single data type of 32-bit values both for signed integer values and memory locations (note that objects and arrays are still annotated with their dynamic type in the heap, as in actual JVM implementations). This differs from most other formalisations where a disjoint set of locations is used (or equivalently, values are tagged with their type), this choice being only informally justified, as *e.g.* by Pusch [14]:

"[. . .] the type information is not used to determine the operational semantics of (correct) JVM code."

Barthe, Dufay, Jakubiec and de Sousa [1] formalized this intuition by considering the actual virtual machine (which is called offensive) as an abstraction of the tagged (defensive) machine, and proving that the former correctly abstracts the latter, whenever the latter does not raise a type error (which is true for verifiable bytecode). Working directly with an untagged semantics immediately frees us from the risk of making unwanted implicit typing assumptions.

A precise model of the memory layout of objects and arrays is however not necessary. It is enough to use functions, state explicitly their domain and not use them outside of it; any concrete representation, for example that maps these domains to sets of offsets, will conform to this model, if the allocator keeps track of the range of objects and does not make them overlap.

Errors We make an important distinction between two kinds of errors:

- Runtime errors that are checked for dynamically and cause the JVM to raise an exception, such as accessing an array out of bounds or putting an element of the wrong type in it, are represented by the absence of transition.
- Actual type errors (called linking errors in the JVM specification) that violate the assumptions that a virtual machine implementation is allowed to make about the code (see [13]), such as dereferencing an integer, or accessing a non-existing field of an object, are represented by a transition to the special state error. This second kind of errors must be correctly handled by the bytecode verification, as the behavior of the virtual machine is unspecified for those cases, and in practice this can result in a crash (in the optimistic case) or the by-passing of access controls.

In the current JVM, the `invokeinterface` instruction raises the exception `IncompatibleClassChangeError` if the receiver of the method does not implement the interface. Because our enhanced bytecode verifier will also typecheck interfaces, we shift this exception from the class of runtime errors to the class of type errors. In our semantics, interface calls are dealt with like virtual calls and it is a type error if the receiver of an interface call does not implement the desired interface. Remark that the runtime errors raised in the explicit cast instruction (which we don't consider) are not removed by this technique.

Objects, arrays and states We write *word* for the set of 32-bit values. Values are used to represent signed integers as well as memory locations. We let fields : $\mathit{class} \to \mathcal{P}(\mathit{fsig})$ be the function that returns the set of (transitively inherited) fields of a class:

$$\text{fields}(C) = \{C.f : t \ \in \mathit{fsig}\} \cup \begin{cases} \text{fields}(\text{super}(C)) & \text{if } C \neq \text{Object} \\ \emptyset & \text{otherwise} \end{cases}$$

An object is a pair $\langle C, o\rangle$ where $C \in \mathit{class}$ and $o : \text{fields}(C) \to \mathit{word}$ gives the value of the relevant fields. We write *object* for the set of objects. We let

array be the set of arrays, annotated with their element type (which can be an array type). We define *heap* as the sets of partial mappings from non-zero values to objects and arrays. The memory allocator is represented by a partial function alloc : $\mathit{heap} \rightharpoonup \mathit{word} \setminus \{0\}$ that maps a heap h to a value that is not defined in h (the absence of value represent the failure of the allocation)[4]: $\forall h \in \mathit{heap} \quad \text{alloc}(h) = v \implies v \notin \text{dom}(h)$. A program state $s = \langle h, l, p \rangle$ consists of a heap, a (total) mapping from variables to values, and a program point.

$$\begin{aligned} \mathit{object} &= \{\langle C, o\rangle \mid C \in \mathit{class}, o : \text{fields}(C) \to \mathit{word}\} \\ \mathit{array} &= \{\langle t, a\rangle \mid t \in \mathit{type}, a : [0, n-1] \to \mathit{word}, n \geq 0\} \\ \mathit{heap} &= \mathit{word} \setminus \{0\} \rightharpoonup (\mathit{object} \cup \mathit{array}) \\ \mathit{state} &= \mathit{heap} \times (\mathit{var} \to \mathit{word}) \times \mathit{ppoint} \end{aligned}$$

Dynamic typing We first recall the standard sub-typing order $\preceq \subseteq \mathit{type} \times \mathit{type}$ induced by the functions super, implements and extends. Note that, in J2SE, every array type is a sub-type of the two interfaces Cloneable and Serializable.

$$\frac{}{t \preceq t} \qquad \frac{t \preceq t' \quad t' \preceq t''}{t \preceq t''} \qquad \frac{t \preceq t'}{t[] \preceq t'[]}$$

$$\frac{}{C \preceq \text{super}(C)} \qquad \frac{I \in \text{implements}(C)}{C \preceq I} \qquad \frac{I' \in \text{extends}(I)}{I \preceq I'}$$

$$\frac{}{I \preceq \text{Object}} \qquad \frac{}{t[] \preceq \text{Cloneable}} \qquad \frac{}{t[] \preceq \text{Serializable}}$$

The key properties that are actually used in the following are i) that $\preceq$ is a partial order ii) the existence of the maximum element (which is called Object in this case) iii) the covariant ordering of array types (third rule in the first line) and of course the link with the functions super, implements and extends, for the language that we consider.

The dynamic typing relation $h \vdash v : t$ between heaps, 32-bit values and types is defined as follows:

$$\frac{}{h \vdash v : \text{int}} \qquad \frac{}{h \vdash 0 : t}$$

$$\frac{h(v) = \langle C, o\rangle \in \mathit{object} \quad C \preceq t}{h \vdash v : t} \qquad \frac{h(v) = \langle t, a\rangle \in \mathit{array} \quad t[] \preceq t'}{h \vdash v : t'}$$

It is worth noting that dynamic types can be checked efficiently by at most a heap access and a sub-typing check. This is important as such an operation is used by the concrete semantics of array assignment.

Method calls Let *bigstep* be the type of big-step semantics for methods.

$$\mathit{bigstep} = \mathcal{P}\left(\bigcup_{n \geq 0} ((\mathit{heap} \times \mathit{word} \times \mathit{word}^n) \times (\mathit{heap} \times \mathit{word} \cup \{\text{error}\}))\right)$$

[4] This is not completely accurate, as the allocation also depends on the needed size.

$$
\begin{array}{lll}
[\![n]\!]_{h,l} & = n & \text{(32-bit signed encoding)} \\
[\![\text{null}]\!]_{h,l} & = 0 & \\
[\![y+z]\!]_{h,l} & = l(y) + l(z) & \\
[\![y.fs]\!]_{h,l} & = \begin{cases} o(fs) & \text{if } h(l(y)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \in \text{fields}(C) \\ \bot & \text{if } l(y) = 0 \\ \text{error} & \text{if } \quad l(y) \neq 0 \ \wedge \ l(y) \notin \text{dom}(h) \\ & \quad \vee \ h(l(y)) \notin \mathit{object} \\ & \quad \vee \ h(l(y)) = \langle C, o\rangle \in \mathit{object} \ \wedge \ \mathit{fs} \notin \text{fields}(C) \end{cases} \\
[\![y[z]]\!]_{h,l} & = \begin{cases} a(l(z)) & \text{if } h(l(y)) = \langle t, a\rangle \in \mathit{array} \ \wedge \ 0 \leq l(z) < |a| \\ \bot & \text{if } l(y) = 0 \ \vee \ h(l(y)) = \langle t, a\rangle \in \mathit{array} \ \wedge \neg \ 0 \leq l(z) < |a| \\ \text{error} & \text{if } l(y) \neq 0 \ \wedge \ l(y) \notin \text{dom}(h) \ \vee \ h(l(y)) \notin \mathit{array} \end{cases}
\end{array}
$$

(a) Semantics $[\![e]\!]_{h,l} \in \mathit{word} \cup \{\text{error}, \bot\}$ of a side-effect free expression e in context h, l

Fig. 2: Semantics of Java bytecode

Let $bs \in \mathit{bigstep}$ and $\langle\langle h, \mathit{this}, \mathit{args}\rangle, r\rangle \in bs$. *this* represents the object on which the method is to be invoked, and *args* represents the list of the arguments. The result r is either the error constant or a pair $\langle h', v\rangle \in \mathit{heap} \times \mathit{word}$ where v is the returned value and h' is the heap obtained by running the method from the initial heap h. The direct method calls that may arise during the execution of the current method are represented by associating a big step transition relation

$$\xrightarrow{ms} \in \mathit{bigstep}$$

to each method signature *ms* (note that the relation for one method signature may correspond to several methods due to dynamic binding). As the relation is supposed to represent every possible call without any assumption on the arguments, is must be defined even for ill-typed ones, possibly with the result error. Also, we make no hypothesis on the correctness of the invoked method yet, thus the error state may be returned even for arguments of the right type. The absence of transition from a particular list of arguments represents non-termination or a runtime exception.

Transition relation The semantics itself is given in Figure 2 by the transition relation

$$\rightarrow \subseteq \mathit{state} \times (\mathit{state} \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$$

A couple of features in this semantics merit explanation: Writing to a field (see Figure 2b) always succeeds (provided the field exists for the target object), even if the value that is written is not of the right type. This is not a safety violation in itself; only a future misuse of this bad value would be an error. Writing to an array always triggers a dynamic check[5] and the execution is stuck if the value stored in the array is not a sub-type of the array's own type, or if the index is out of bounds. Remember that a run-time exception of virtual

[5] This is unavoidable with the covariant arrays of the Java type system.

$$
\begin{aligned}
[\![x := e]\!](h,l) &= \left\{\begin{array}{ll} h, l[x \leftarrow [\![e]\!]_{h,l}] & \\ \quad \text{if } [\![e]\!]_{h,l} \notin \{\bot, \text{error}\} & \\ \bot \quad \text{if } [\![e]\!]_{h,l} = \bot & \\ \text{error if } [\![e]\!]_{h,l} = \text{error} & \end{array}\right\} \quad \begin{array}{l}\text{if } e \notin \{y.ms(\cdots),\\ \qquad \mathsf{new}\ C,\\ \qquad \mathsf{new}\ t[y]\}\end{array}\\
[\![x := \mathsf{new}\ C]\!](h,l) &= \left\{\begin{array}{ll} h[v \leftarrow \langle C, \lambda fs \in \text{fields}(C).0\rangle], l[x \leftarrow v] & \\ \quad \text{if alloc}(h) = v & \\ \bot \quad \text{if } h \notin \text{dom(alloc)} & \end{array}\right.\\
[\![x := \mathsf{new}\ t[y]]\!](h,l) &= \left\{\begin{array}{ll} h[v \leftarrow \langle t, \lambda i \in [0, l(y)-1].0\rangle], l[x \leftarrow v] & \\ \quad \text{if } l(y) \geq 0 \ \wedge\ \mathit{alloc}(h) = v & \\ \bot \quad \text{if } l(y) < 0 \ \vee\ h \notin \text{dom(alloc)} & \end{array}\right.\\
[\![x.fs := y]\!](h,l) &= \left\{\begin{array}{ll} h[l(x) \leftarrow \langle C, o[fs \leftarrow l(y)]\rangle], l & \\ \quad \text{if } h(l(x)) = \langle C, o\rangle \in \mathit{object} \ \wedge\ fs \in \text{fields}(C) & \\ \bot \quad \text{if } l(x) = 0 & \\ \text{error if} \quad l(x) \neq 0 \ \wedge\ l(x) \notin \text{dom}(h) & \\ \qquad \vee\ h(l(x)) \notin \mathit{object} & \\ \qquad \vee\ h(l(x)) = \langle C, o\rangle \in \mathit{object} \ \wedge\ fs \notin \text{fields}(C) & \end{array}\right.\\
[\![x[y] := z]\!](h,l) &= \left\{\begin{array}{ll} h[l(x) \leftarrow \langle t, a[l(y) \leftarrow l(z)]\rangle], l & \\ \quad \text{if} \quad h(l(x)) = \langle t, a\rangle \in \mathit{array} & \\ \qquad \wedge\ h \vdash l(z) : t \wedge\ 0 \leq l(y) < |a| & \\ \bot \quad \text{if} \quad l(x) = 0 & \\ \qquad \vee \quad l(x) = \langle t, a\rangle \in \mathit{array} & \\ \qquad\quad \wedge\ \neg\,(h \vdash l(z) : t \ \wedge\ 0 \leq l(y) < |a|) & \\ \text{error if} \quad l(x) \neq 0 \ \wedge\ l(x) \notin \text{dom}(h) \ \vee\ h(l(x)) \notin \mathit{array} & \end{array}\right.
\end{aligned}
$$

(b) Semantics $[\![i]\!] : \mathit{heap} \times (\mathit{var} \to \mathit{word}) \to (\mathit{heap} \times (\mathit{var} \to \mathit{word}) \cup \{\text{error}, \bot\})$ of a non-branching intraprocedural instruction i

$$
\frac{i \notin \{\mathsf{goto}\ p, \mathsf{if} \cdots, \mathsf{return}\ x, x := y.ms(\cdots)\} \quad [\![i]\!](h,l) \neq \bot}{h, l \xrightarrow{i} [\![i]\!](h,l)} \qquad \frac{}{h, l \xrightarrow{\mathsf{return}\ x} h, l(x)}
$$

$$
\frac{h, l(y), l(x_1), \ldots, l(x_n) \xrightarrow{ms} h', v}{h, l \xrightarrow{x := y.ms(x_1,\ldots,x_n)} h', l[x \leftarrow v]} \qquad \frac{h, l(y), l(x_1), \ldots, l(x_n) \xrightarrow{ms} \text{error}}{h, l \xrightarrow{x := y.ms(x_1,\ldots,x_n)} \text{error}}
$$

(c) Semantics $\xrightarrow{i} \subseteq (\mathit{heap} \times (\mathit{var} \to \mathit{word})) \times (\mathit{heap} \times (\mathit{var} \to \mathit{word}) \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$ of a non-branching instruction i

$$
\frac{\text{code}(p) \notin \{\mathsf{goto}\ p', \mathsf{if} \cdots\} \quad h, l \xrightarrow{\text{code}(p)} r \quad r \in \mathit{heap} \times \mathit{word} \cup \{\text{error}\}}{\langle h, l, p\rangle \to r}
$$

$$
\frac{\text{code}(p) \notin \{\mathsf{goto}\ p', \mathsf{if} \cdots\} \quad h, l \xrightarrow{\text{code}(p)} h', l'}{\langle h, l, p\rangle \to \langle h', l', p+1\rangle} \qquad \frac{\text{code}(p) = \mathsf{goto}\ p'}{\langle h, l, p\rangle \to \langle h, l, p'\rangle}
$$

$$
\frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p' \quad l(x) < l(y)}{\langle h, l, p\rangle \to \langle h, l, p'\rangle} \qquad \frac{\text{code}(p) = \mathsf{if}\ x < y\ \ p' \quad l(x) \geq l(y)}{\langle h, l, p\rangle \to \langle h, l, p+1\rangle}
$$

(d) Small step transition relation $\to \subseteq \mathit{state} \times (\mathit{state} \cup \mathit{heap} \times \mathit{word} \cup \{\text{error}\})$

Fig. 2: Semantics of Java bytecode (continued)

machine is modelled in our setting by the execution being stuck. Finally, in Figure 2c, it is important to remember that a method call can occur with ill-typed arguments, and that the invoked method itself can be ill-typed, hence the rule for propagating the error state.

2.3 Memory safety

We give a modular definition of memory safety that is stronger than what is actually needed for a complete program: it includes the preservation of the well-typedness of the heap, and the fact that the heap is only extended, which is needed to ensure a global safety. This property requires some prior definitions to express those accurate invariants about the heap.

Well typed heaps The following relation expresses that a heap is consistent.

$$\frac{\forall v \in \mathit{word} \setminus \{0\} \quad \begin{cases} & v \notin \mathrm{dom}(h) \\ \vee & h(v) = \langle C, o\rangle \in \mathit{object} \\ & \wedge\ \forall C'.f : t \in \mathrm{fields}(C) \quad h \vdash o(C'.f : t) : t \\ \vee & h(v) = \langle t, a\rangle \in \mathit{array} \\ & \wedge\ \forall i \in [0, |a| - 1] \quad h \vdash a(i) : t \end{cases}}{h \vdash h}$$

Ordering on heaps The relation $\Subset$ expresses the preservation of existing objects between two heaps.

$$\frac{\forall v \in \mathit{word} \setminus \{0\} \quad \begin{cases} & v \notin \mathrm{dom}(h) \\ \vee & h(v) = \langle C, o\rangle \in \mathit{object} \ \wedge h'(v) = \langle C, o'\rangle \in \mathit{object} \\ \vee & h(v) = \langle t, a\rangle \in \mathit{array} \ \wedge h'(v) = \langle t, a'\rangle \in \mathit{array} \end{cases}}{h \Subset h'}$$

Modular memory-safety Definition 1 introduces the general safety property for the transition relation associated with a method. We need to give two variants of it since we use slightly different formalisations for the transition relation of the current method and the relations representing method calls. As errors are immediately propagated in the semantics, it is sound to define the safety as the unreachability of error in the outermost invocation.

Definition 1. *The relation* $\rightarrow \subseteq \mathit{state} \times (\mathit{state} \cup (\mathit{heap} \times \mathit{word}) \cup \{\mathrm{error}\})$ *is safe with respect to* $\mathrm{t}_{\mathrm{arg}}$ *and* $\mathrm{t}_{\mathrm{ret}}$ *if for all* h, l *such that* $h \vdash h \ \wedge\ \forall x \in \mathit{arg} \quad h \vdash l(x) : \mathrm{t}_{\mathrm{arg}}(x)$ *then* $\langle h, l, 0\rangle \not\rightarrow^* \mathrm{error}$ *and* $\langle h, l, 0\rangle \rightarrow^* h', v \implies h \Subset h' \ \wedge\ h' \vdash h' \ \wedge\ h' \vdash v : \mathrm{t}_{\mathrm{ret}}$.

Similarly, a transition relation $\xrightarrow{ms} \in \mathit{bigstep}$ *is safe with respect to the signature* $ms = t.m(t_1, \ldots, t_n)$ *if for all* $h, v, v_1, \ldots, v_n$ *such that* $h \vdash h \ \wedge\ h \vdash v : t \ \wedge\ \forall i \leq n \quad h \vdash v_i : t_i$ *then* $h, v, v_1, \ldots, v_n \not\xrightarrow{ms} \mathrm{error}$ *and* $h, v, v_1, \ldots, v_n \xrightarrow{ms} h', v' \implies h \Subset h' \ \wedge\ h' \vdash h' \ \wedge\ h' \vdash v' : \mathrm{result}(ms)$.

Memory safety and security In this paper we focus on memory safety which is just one aspect of the security of Java bytecode. Memory safety ensures that the virtual machine will not crash when executing the program but the secure execution of untrusted bytecode requires stronger properties. A common (informal) security requirement is that a program should not be able to forge a pointer to a heap location that it is not supposed to have access to; this is not easy to define formally without instrumenting the semantics. With the semantics defined in this section we can prove that a method whose return type is a reference type will return a heap location that is either unnallocated in the initial heap or reachable through the (reference) arguments with which the method was invoked. This is ensured by the analysis of Section 3 without any modification: only the proofs must be extended by strengthening the concretisation function for the abstract domain. This is still too simplistic, as it does not distinguish between private or public fields, nor does it account for the fact that an untrusted program can be given controlled access to some (private) objects through the invocation of trusted methods from the API. Nevertheless, even though the memory safety defined here does not in itself imply any access restriction property, the analysis by which we ensure memory safety represents a large part of what is needed to ensure security, as shown by Leroy and Rouaix [12] who formalize such stronger security properties and give sufficient conditions, in addition to well-typedness, for an applet to be safe with respect to these properties.

3 Extended bytecode typing

In this section we present an extended abstract domain for bytecode verification and prove it sound with respect to the semantics. The main difference with the standard verifier is the use of interfaces in types, which make the runtime check in the "invokeinterface" instruction unnecessary. Another difference is that integers are abstracted by $\top_v$. This simplifies the presentation and the stack maps.

3.1 Abstract domain

The abstract domain elements are called *stack maps* in the context of Java bytecode verification, as they normally map program points to abstract operand stacks. We keep this name even though the stack is absent in our setting. Our abstract domain associates a type to each variable at each program point. This type is either $\top_v$ (for non-reference values), null, or a conjunction of object (or array) types.

$$
\begin{array}{lcl}
\mathit{value}^\sharp & ::= & \mathrm{null} \quad | \quad \top_v \\
& & \quad | \; t_1 \wedge \cdots \wedge t_n \quad n \geq 1, t_i \in \mathit{type} \setminus \{\mathrm{int}\}, \\
& & \qquad\qquad\qquad\quad \forall i, j \leq n \quad t_i \preceq t_j \implies i = j \\
\mathit{state}^\sharp & = & \mathit{var} \to \mathit{value}^\sharp \\
\mathit{map} & = & \mathit{ppoint} \to \mathit{state}^\sharp \\
\\
{\mathit{state}^\sharp}_{\bot_s}^{\top_s} & = & \mathit{state}^\sharp \cup \{\bot_s, \top_s\}
\end{array}
$$

The two special abstract states $\bot_s$ and $\top_s$ indicate respectively an unreachable program point and the possibility of the (concrete) error state being reachable. We also define the abbreviation $state^\sharp{}_{\bot_s}^{\top_s}$. Conjunctions are defined up to the order, *i.e.*, $t \wedge t' = t' \wedge t$.

A conjunction is to be interpreted as the set of objects that are a member of every atomic type in it. Note that we only consider conjunctions of unordered atomic types. This is necessary to be able to define an order on abstract values and not just a pre-order, and also to make the concretisation (almost) injective (as adding a super-type of another conjunct does not change the concretisation of a conjunction). Another isomorphic solution is to consider upward-closed (with respect to $\preceq$) sets of atomic types. We choose the first representation which is more compact and hence allows us to compute least upper bounds efficiently (see below).

The following definition identifies a subset of stack maps in which we want to choose a "certificate" to send with the current method.

Definition 2. *A stack map $m \in map$ is conjunction-free if all of its conjunctions $t_1 \wedge \cdots \wedge t_n$ are reduced to one element (*i.e.*, $n = 1$).*

Concretisation We define the concretisation functions $\gamma_h : value^\sharp \to \mathcal{P}(word)$, $h \in heap$, $\gamma_p : state^\sharp{}_{\bot_s}^{\top_s} \to \mathcal{P}(state \cup \{\text{error}\})$, $p \in ppoint$ and $\gamma : (value^\sharp \cup \{\top_s\}) \to \mathcal{P}((heap \times word) \cup \{\text{error}\})$ by

$$
\begin{array}{ll}
\gamma_h(\top_v) & = word \\
\gamma_h(\text{null}) & = \{0\} \\
\gamma_h(t_1 \wedge \cdots \wedge t_n) & = \{v \in word \mid \forall i \leq n \quad h \vdash v : t_i\}
\end{array}
$$

$$
\begin{array}{ll}
\gamma_p(\bot_s) & = \emptyset \\
\gamma_p(\top_s) & = state \cup \{\text{error}\} \\
\gamma_p(l^\sharp) & = \{\langle h, l, p\rangle \in state \mid \ h \vdash h \ \wedge \ \forall x \in var \quad l(x) \in \gamma_h(l^\sharp(x))\}
\end{array}
$$

$$
\begin{array}{ll}
\gamma(\top_s) & = heap \times word \cup \{\text{error}\} \\
\gamma(v^\sharp) & = \left\{h, v \mid h \vdash h \ \wedge \ v \in \gamma_h(v^\sharp)\right\}.
\end{array}
$$

Partial order, least upper bound and transfer function The partial orders $\sqsubseteq_v \subseteq value^\sharp \times value^\sharp$ and $\sqsubseteq \subseteq state^\sharp{}_{\bot_s}^{\top_s} \times state^\sharp{}_{\bot_s}^{\top_s}$ are defined by the following rules:

$$
\frac{}{\text{null} \sqsubseteq_v v^\sharp} \qquad \frac{}{v^\sharp \sqsubseteq_v \top_v} \qquad \frac{\forall j \leq n' \quad \exists i \leq n \quad t_i \preceq t'_j}{t_1 \wedge \cdots \wedge t_n \ \sqsubseteq_v \ t'_1 \wedge \cdots \wedge t'_{n'}}
$$

$$
\frac{}{\bot_s \sqsubseteq s^\sharp} \qquad \frac{}{s^\sharp \sqsubseteq \top_s} \qquad \frac{\forall x \in var \quad l^\sharp(x) \sqsubseteq l'^\sharp(x)}{l^\sharp \sqsubseteq l'^\sharp}
$$

The (commutative) least upper bound operators $\sqcup_v : \mathit{value}^\sharp \times \mathit{value}^\sharp \to \mathit{value}^\sharp$ and $\sqcup : {\mathit{state}^\sharp}^{\top_s}_{\bot_s} \times {\mathit{state}^\sharp}^{\top_s}_{\bot_s} \to {\mathit{state}^\sharp}^{\top_s}_{\bot_s}$ are defined by

$$\begin{array}{ll}
\mathrm{null} \sqcup_v v^\sharp & = v^\sharp \\
v^\sharp \sqcup_v \top_v & = \top_v \\
t_1 \wedge \cdots \wedge t_n \\
\sqcup_v\ t'_1 \wedge \cdots \wedge t'_{n'} & = \begin{cases} \text{let } \mathcal{T} = \{t \in \mathit{type} \mid \exists i \leq n, j \leq n' \quad t_i \preceq t \ \wedge\ t'_j \preceq t\} \\ \text{in } \bigwedge\{t \in \mathcal{T} \mid \forall t' \in \mathcal{T} \quad t' \preceq t \implies t' = t\} \end{cases}
\end{array}$$

$$\begin{array}{ll}
\bot_s \sqcup s^\sharp & = s^\sharp \\
s^\sharp \sqcup \top_s & = \top_s \\
l^\sharp \sqcup l'^\sharp & = \lambda x. l^\sharp(x) \sqcup_v l'^\sharp(x).
\end{array}$$

The least upper bound of two (non-empty) conjunctions is always defined (and non-empty), because Object is a super type of all reference types (including interfaces and array types). The second line in the least upper bounds of two conjunctions ensures that we keep only maximal atoms. The actual computation of the least upper bound of two conjunctions can be performed efficiently: as only minimal types $t \in \mathcal{T}$ will be kept, it is sufficient to find the first super-class and/or super-interfaces of each pair t_i, t'_j, which is done by traversing the hierarchy above t_i and t'_j. Figure 3 defines the abstract semantics as two relations:

$$\begin{array}{l}
\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to {\mathit{state}^\sharp}^{\top_s}_{\bot_s}) \times \mathit{ppoint} \\
\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{value}^\sharp)
\end{array}$$

3.2 Correctness of the abstraction

Lemma 1 ensures the consistency of the partial order with respect to the concretisation, which is crucial for the correctness of the verification.

Lemma 1. *For all $h \in \mathit{heap}$ and $v^\sharp, v'^\sharp \in \mathit{value}^\sharp$, if $v^\sharp \sqsubseteq_v v'^\sharp$ then $\gamma_h(v^\sharp) \subseteq \gamma_h(v'^\sharp)$. For all $p \in \mathit{ppoint}$ and $s^\sharp, s'^\sharp \in {\mathit{state}^\sharp}^{\top_s}_{\bot_s}$, if $s^\sharp \sqsubseteq s'^\sharp$ then $\gamma_p(s^\sharp) \subseteq \gamma_p(s'^\sharp)$. For all $v^\sharp, v'^\sharp \in \mathit{value}^\sharp \cup \{\top_s\}$, if $v'^\sharp = \top_s \vee v^\sharp \sqsubseteq_v v'^\sharp$ then $\gamma(v^\sharp) \subseteq \gamma(v'^\sharp)$.*

The least upper bound operator is used during the fix-point computation and must be correct for the analyser to be correct (but not for the checker).

Lemma 2. *For all $v^\sharp, v'^\sharp \in \mathit{value}^\sharp$, $v^\sharp \sqcup_v v'^\sharp \sqsupseteq_v v^\sharp$ and $v^\sharp \sqcup_v v'^\sharp \sqsupseteq_v v'^\sharp$. For all $s^\sharp, s'^\sharp \in {\mathit{state}^\sharp}^{\top_s}_{\bot_s}$, $s^\sharp \sqcup s'^\sharp \sqsupseteq s^\sharp$ and $s^\sharp \sqcup s'^\sharp \sqsupseteq s'^\sharp$.*

The core of the correctness of the checker (and of the analyser) resides in Lemma 3 which says that the concrete transition relation is correctly approximated.

Lemma 3. *Suppose that for every signature ms the relation $\xrightarrow{ms}$ is safe with respect to ms (see Definition 1). Let $s \in \mathit{state}$, $r \in \mathit{state} \cup (\mathit{heap} \times \mathit{word}) \cup \{\mathrm{error}\}$, $p \in \mathit{ppoint}$ and $l^\sharp \in \mathit{state}^\sharp$ such that $s \to r$ and $s \in \gamma_p(l^\sharp)$. Then one of the following holds:*

$$
\begin{array}{ll}
[\![n]\!]^\sharp_{l^\sharp} & = \top_v \\
[\![\text{null}]\!]^\sharp_{l^\sharp} & = \text{null} \\
[\![y + z]\!]^\sharp_{l^\sharp} & = \top_v \\
[\![\mathsf{new}\ C]\!]^\sharp_{l^\sharp} & = C \\
[\![\mathsf{new}\ t[y]]\!]^\sharp_{l^\sharp} & = t[] \\
[\![y.fs]\!]^\sharp_{l^\sharp} & = \begin{cases} t & \text{if } fs = C.f : t \ \wedge\ l^\sharp(y) \sqsubseteq_v C \\ \top_s \text{ otherwise} \end{cases} \\
[\![y[z]]\!]^\sharp_{l^\sharp} & = \begin{cases} t & \text{if } l^\sharp(y) = t[] \quad t \in \mathit{type} \\ \bot_s \text{ if } l^\sharp(y) = \text{null} \\ \top_s \text{ if } l^\sharp(y) \notin \{\text{null}\} \cup \{t[] \mid t \in \mathit{type}\} \end{cases} \\
[\![y.ms(x_1, \ldots, x_n)]\!]^\sharp_{l^\sharp} & = \begin{cases} \text{result}(ms) \\ \quad \text{if} \quad ms = t.m(t_1, \ldots, t_n) \\ \quad\quad \wedge\ l^\sharp(y) \sqsubseteq_v t \ \wedge\ \forall i \leq n \quad l^\sharp(x_i) \sqsubseteq_v t_i \\ \top_s \text{ otherwise} \end{cases}
\end{array}
$$

(a) Abstract semantics $[\![e]\!]^\sharp_{l^\sharp} \in \mathit{value}^\sharp \cup \{\bot_s, \top_s\}$ of an expression e in the abstract context $l^\sharp$

$$
\begin{array}{ll}
[\![x := e]\!]^\sharp(l^\sharp) & = \begin{cases} l^\sharp[x \leftarrow [\![e]\!]^\sharp_{l^\sharp}] \\ \quad \text{if } [\![e]\!]^\sharp_{l^\sharp} \notin \{\bot_s, \top_s\} \\ \bot_s \text{ if } [\![e]\!]^\sharp_{l^\sharp} = \bot_s \\ \top_s \text{ if } [\![e]\!]^\sharp_{l^\sharp} = \top_s \end{cases} \\
[\![x.fs := y]\!]^\sharp(l^\sharp) & = \begin{cases} l^\sharp \quad \text{if} \quad fs = C.f : t \\ \quad\quad \wedge\ l^\sharp(x) \sqsubseteq_v C \ \wedge\ l^\sharp(y) \sqsubseteq_v t \\ \top_s \text{ otherwise} \end{cases} \\
[\![x[y] := z]\!]^\sharp(l^\sharp) & = \begin{cases} l^\sharp \quad \text{if } l^\sharp(x) = t[] \quad t \in \mathit{type} \\ \bot_s \text{ if } l^\sharp(x) = \text{null} \\ \top_s \text{ if } l^\sharp(y) \notin \{\text{null}\} \cup \{t[] \mid t \in \mathit{type}\} \end{cases}
\end{array}
$$

(b) Abstract semantics $[\![i]\!]^\sharp : \mathit{state}^\sharp \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}$ of a non-branching instruction i ($i \neq \mathsf{return}\ x$)

$$
\frac{\text{code}(p) \notin \{\mathsf{return}\ x, \mathsf{goto}\ p', \mathsf{if} \cdots\}}{p \overset{[\![\text{code}(p)]\!]^\sharp}{\longrightarrow} p + 1} \qquad \frac{\text{code}(p) = \mathsf{goto}\ p'}{p \overset{\lambda l^\sharp . l^\sharp}{\longrightarrow} p'}
$$

$$
\frac{\text{code}(p) = \mathsf{if}\ x < y \quad p'}{p \overset{\lambda l^\sharp . l^\sharp}{\longrightarrow} p'} \qquad \frac{\text{code}(p) = \mathsf{if}\ x < y \quad p'}{p \overset{\lambda l^\sharp . l^\sharp}{\longrightarrow} p + 1}
$$

$$
\frac{\text{code}(p) = \mathsf{return}\ x}{p \overset{\lambda l^\sharp . l^\sharp(x)}{\longrightarrow}}
$$

(c) Abstract transition relations $\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{state}^{\sharp\,\top_s}_{\bot_s}) \times \mathit{ppoint}$ and $\longrightarrow \subseteq \mathit{ppoint} \times (\mathit{state}^\sharp \to \mathit{value}^\sharp)$

Fig. 3: Abstract semantics of Java bytecode

1. $p \xrightarrow{F^\sharp} p'$ *and* $r \in \gamma_{p'}(F^\sharp(l^\sharp))$ *for some* p' *and* $F^\sharp$, *or*
2. $p \xrightarrow{F^\sharp}$ *and* $r \in \gamma(F^\sharp(l^\sharp))$ *for some* $F^\sharp$.

Furthermore, the functions $F^\sharp$ *are monotone.*

Formal proofs can be found in the technical report [2].

3.3 Analysis and checking

The following definition introduce the notion of witness of the current method whose signature is given by $\mathrm{t}_{\mathrm{arg}}$ and $\mathrm{t}_{\mathrm{ret}}$.

Definition 3. *A witness is a stack map* $m \in map$ *such that*

1. $\forall x \in var \quad m(0)(x) \sqsupseteq_v \begin{cases} \mathrm{t}_{\mathrm{arg}}(x) & \text{if } x \in arg \\ \top_v & \text{otherwise} \end{cases}$
2. $\forall p, p' \in ppoint \quad p \xrightarrow{F^\sharp} p' \implies F^\sharp(m(p)) \sqsubseteq m(p')$
3. $\forall p \in ppoint \quad p \xrightarrow{F^\sharp} \implies F^\sharp(m(p)) \sqsubseteq_v \mathrm{t}_{\mathrm{ret}}$

Note that by definition of stack maps, witnesses contain no $\top_s$ or $\bot_s$. This correspond respectively to the assumptions that the code should type without error, and that even dead code should be typable. This second condition is necessary for the pruning to work.

The following lemma shows that the memory safety property can be ensured by simply checking that some given stack map is a witness, and shows how to compute the least witness. The next section will show that, for Java programs, the least witness can be pruned resulting in a witness without conjunction.

Theorem 1. *Suppose that every relation* $\xrightarrow{ms}$ *is safe with respect to ms (see Definition 1). If there exists a witness m then the relation* $\rightarrow$ *is safe with respect to* $\mathrm{t}_{\mathrm{arg}}$ *and* $\mathrm{t}_{\mathrm{ret}}$. *Moreover, the least witness*[6] *(if there exists a witness) can be computed by fixpoint iteration.*

The proof can be found in the technical report [2].

4 Lightweight verification by fix-point pruning

In the previous section we formalised a bytecode analysis extended to interfaces, using conjunctions of types in the abstract domain. The drawback of this extension is its computational cost, especially in terms of memory, that could make it unapplicable on the smallest Java capable devices. We will now present an additional step to the lightweight verification setting that removes the need for computations of sets of types on the consumer side by computing a witness without conjunction, if the safety of the program does not rely on them. This is the case for programs compiled from Java in particular.

[6] The abstract state $\bot_s$ is not a witness by definition, but the state $\lambda x \in var$.null is a minimum of $state^\sharp$. Thus, if the set of witnesses is not empty, it has a least element.

```
public static void main(String [] args) {
    I3 ∧ I4 i = args.length > 0 ? new A() : new B();
    i.m3();
    i.m4();
}
```

Fig. 4: A safe Java bytecode program that has no conjunction-free witness

4.1 Stack map checking without conjunctions

We first present an algorithm, which when possible, computes a conjunction-free witness from the least fix-point. The key hypothesis of this algorithm is therefore the existence of a witness without conjunction. This is not the case of all bytecode programs. Figure 4 shows a method that has no such witness (`I3`, `I4`, `A` and `B` are defined as in Section 1). It is shown in pseudo Java (with a conjunction of types) but has to be written directly in bytecode.

As explained in the previous section, the checking of bytecode mainly consist in the verification of the conditions of Definition 3, which reduces to computations of the functions $F^\sharp$ of the abstract semantics, and ordering checks $\sqsubseteq$ between abstract states. As the $F^\sharp$s never "create" a conjunction of types (this is easily verified on the definition), we can see that the checking of a conjunction-free witness can be performed without manipulating conjunctions[7]. So the technique of pruning removes the need for conjunction computations.

Fix-point pruning The algorithm of Figure 5 optimistically searches for such a witness. It starts from the least witness (if it exists)

$$\text{lfp} \in \mathit{map}$$

computed by direct analysis, and traverses the set of conjunction-free stack maps that are greater than lfp, until a witness is reached or the search space is exhausted. If there exists a witness without conjunction, it must belong to this set, by definition of the least fix-point. Moreover, the finite ascending chain condition satisfied by the lattice ensures that the search space is finite so the exploration terminates (which is interesting in case there is no such witness). Therefore this algorithm is complete in the sense that if a solution exists, it will find it.

More precisely, the idea is to start from $\top_v$ at each program point and each variable and replace those values by lesser ones until a witness is reached. The non-deterministic instruction "choose" is to be interpreted as follows: if the choice fails at any point (*i.e.*, there is no $v^\sharp$ satisfying the required conditions) then we backtrack to a previous choice. The algorithm can terminate either by returning a witness, or by returning nothing, if every combination of choices

[7] In the real process, the value of the witness is only sent for some program points and the remaining values are reconstructed at checking time, but no least upper bound is involved, thus the property still holds.

let $w = \lambda p \in ppoint.\lambda x \in var.\top_v$ and $W = ppoint$
while $W \neq \emptyset$ do
take $p \in W$ (and remove it)
choose a maximal $l^\sharp \in state^\sharp$ such that
$l^\sharp$ is without conjunction and $\mathrm{lfp}(p) \sqsubseteq l^\sharp \sqsubseteq w(p)$
and $\forall p' \in ppoint \quad p \xrightarrow{F^\sharp} p' \implies$
$F^\sharp(l^\sharp) \neq \top_s$
$\wedge\ F^\sharp(l^\sharp) \sqsubseteq w(p')$
$\wedge\, p \xrightarrow{F^\sharp} \implies F^\sharp(l^\sharp) \sqsubseteq_v \mathrm{t_{ret}}$
if $l^\sharp \neq w(p)$ then
$w := w[p \leftarrow l^\sharp]$
$W := W \cup \{p' \mid p' \xrightarrow{F^\sharp} p\}$
end if
done
return w

Fig. 5: Naive (complete) pruning algorithm

eventually gets stuck. As for the strategy used to implement the work-set (instruction "take"), we found that a stack without duplicates was an efficient and simple heuristic. Note that this second sort of choice is never undone and does not cause further backtracking.

The following theorem formalizes the fact that the algorithm of Figure 5 is sound and complete (when a witness without conjunction exists).

Theorem 2. *The complete pruning algorithm always terminates, either by returning a stack map w or by a failure in the choice of $l^\sharp$. If it terminates by returning some w, then w is a conjunction-free witness. Furthermore, if a conjunction-free witness w exists, the algorithm will return such a witness.*

The proof can be found in the technical report [2].

Java programs In the Java language, all variables are declared with a fixed type $t \in type$ (actually, the basic types are not exactly the same between Java and bytecode: the smaller integer types are merged with int in the latter). This type satisfies the same constraints that are expressed by the abstract semantics in the previous section (including the ones for interfaces) and can therefore be considered as a witness for each method, where the type of every variable is the same regardless of the program point. The difference is that the variables are the source variables, not the bytecode local variables and stack positions.

If the compiler does not transform the structure of the program too much, more precisely if each variable of the source program is mapped to a (bytecode) local variable in a given subset of the (bytecode) program points, without overlapping, then we see that the witness representing the typing of the source code can be renamed to a corresponding witness on the bytecode. This witness has an

```
public static void main(String [] args) {
    I3 i3 = args.length > 0 ? new A() : new B();
    I4 i4 = args.length > 0 ? new A() : new B();
    i3.m3();
    i4.m4();
    Object i = args.length > 0 ? i3 : i4;
    i.toString();
}
```

Fig. 6: A Java program for which a conjunction-free witness cannot be build from the atomic types of the least fix-point.

interesting feature: it does not contain any conjunction (because variables are declared with a single type). Therefore, for bytecode compiled from Java with a "natural" compiler, there exists a conjunction-free witness for every method (and thus the algorithm of the previous section will find it).

An alternative solution for introducing the verification of interfaces in a lightweight verification process in the case of Java (source) programs is to generate a stack map from the type annotations present in the source code. Indeed, as the lightweight verification paradigm is being generalized to J2SE Java [9], the task of generating stack maps is moving from a dedicated "preverifier" program to the compiler itself. One disadvantage of this technique is that all the tools that manipulate bytecode (notably the compiler) must take care of stack maps consistently, which complicates their design. Instead, we extract a witness without conjunction directly from the bytecode (given that there exists one).

4.2 Efficient fix-point pruning

Although the first algorithm is complete, it takes potentially a very long time, since the search space is the product over program points and local variables of the part of the type hierarchy that is greater than the corresponding value type in the least fix-point. In fact, it is rarely necessary, for a given variable and program point, to consider the entire type hierarchy above the type given by the least fix-point. Most of the time it is enough to choose one of its conjuncts (if it is a conjunction). The resulting pruning algorithm still has an exponential complexity, but it performs reasonably fast in practice.

Reducing the branching factor In most cases, the following holds: there exists a witness $\mathrm{w} \in \mathit{map}$ without conjunction such that the atomic types that appear in w are atoms of the corresponding conjunctions in the least fix point.

$$\forall p \in \mathit{ppoint} \quad \forall x \in \mathit{var} \quad \mathrm{w}(p)(x) = t \in \mathit{type} \implies \mathrm{lfp}(p)(x) = t \ \wedge \cdots$$

This is not true for the program in Figure 6 (`I3`, `I4`, `A` and `B` are defined as in Section 1). In this example, we build two variables `i3` and `i4` with most precise

type `I3` $\wedge$ `I4`. The variable `i` is then defined as the "union" of the two, and its type is therefore `I3` $\wedge$ `I4`. However, in a stack map without conjunction, the type of `i3` must be `I3`, and the type of `i4` must be `I4` (because we call `m3` and `m4`, respectively). Therefore, the type of `i` must be greater than the least upper bound of `I3` and `I4`, *i.e.*, `java.lang.Object`, which is not an atom of `I3` $\wedge$ `I4`. Nevertheless, this seems to be a pathological example and in practise the above hypothesis holds for all our (substantial) benchmarks, which indicates that it's applicability is very general.

Algorithm Taking into account the hypothesis of section 4.2, we proceed by searching for a witness satisfying this hypothesis. The optimized algorithm is obtained by replacing the condition

$$l^\sharp \text{ is without conjunction and } \mathrm{lfp} \sqsubseteq l^\sharp$$

by the stronger condition

$$\begin{aligned} \forall x \in var \qquad & l^\sharp(x) = \top \\ & \vee\ l^\sharp(x) = t \in \mathit{type}\ \wedge\ \mathrm{lfp}(p)(x) = t\ \wedge \cdots \\ & \vee\ l^\sharp(x) = \mathrm{null} = \mathrm{lfp}(p)(x) \end{aligned}$$

in the algorithm of Figure 5.

Correctness As the complete version, the new algorithm is sound and terminates and, though it is incomplete (see the above counter-example), it will succeed in finding a witness if the optimistic hypothesis holds. If not, the search will fail and the complete algorithm presented before should then be run instead.

5 Experiments

We have implemented our ideas in a verifier for real Java bytecode that adds the stack maps as method attributes, as defined in the JVM specification.

5.1 Extensions and limitations

The bytecode language presented so far is considerably simplified. We had to address a few more issues to deal with real bytecode. First, the Java bytecode uses an operand stack in addition to local variables. This complicates the abstract states, as they now have another list of abstract values, of variable length. Of course, this adds more reasons for the verification to fail, namely, (operand) stack overflow or underflow, or the possibility to have different stack heights at some program point. Second, in addition to 32-bit integers, Java bytecode has floats, longs and doubles. floats are easily abstracted by $\top_v$, and the 64-bit types by two $\top_v$s. Note that, although 32-bit integers are not distinguished from shorter integers at the bytecode level (they are in the source code), this is not the case for arrays of such types. Therefore, to ensure that the array bounds checks

performed at runtime correctly interpret the length field, the size of elements must be known. This implies that arrays of floats or ints must not be confused with arrays of shorts. However, individual float and int values can still be merged, since the instruction for accessing arrays are typed. The last additional feature is throwing and catching exceptions. From the verification point of view, exceptions just add some more transitions in the control flow graph, with a semantics that empties the stack and then push some constant reference type (the type that is caught by the corresponding handler). They pose no particular difficulty.

While analysing real bytecode, we have omitted some aspects of the verification in the concrete implementation. In particular, we do not address the issue of verifying sub-routines. Also, a byte code verifier should verify that any object is initialized before being used. The benchmarks presented here do not include this phase. Finally, note that the semantics that we gave to the bytecode used big-step calls, which prevents us to consider even a simplified (interleaving) version of concurrency, which would require explicit call stacks. Therefore, in principle, all the results presented here only applies to single-threaded programs. However, the scheme of the proof (an invariant that holds at each state of a small-step semantics) does not seem to rely on sequentiality, and we believe that there is no issue in extending it to threads.

5.2 Stack maps and checking

The stack maps that are attached to the bytecode are not exactly a representation of conjunction-free witnesses, but only of the value of such witnesses for a subset of the program points. These points correspond basically to the basic blocks of the control flow graph. This reduces the size of stack maps, while still allowing a very simple checking algorithm that evaluates program points in order. We will not detail this aspect, as we used the same subset of program points and the same checking algorithm as Sun's lightweight bytecode verifier. The resulting stack maps are encoded in the class files either as StackMap attributes in the same format as the lightweight bytecode verifier, or with a new attribute using a sparse representation. In the latter, we just replaced an array of value types by an array of bits (to indicate which values are not $\top_v$) followed by the list of non-$\top_v$ values. In order for the comparison to be fair, the sparse representation uses the same verbose encoding of value types as the stack map representation.

5.3 Results

Three test suites were used to experiment with the analysis, pruning and checking with interfaces. The first one consists of 433 old midlets (Java applets for mobile phones) downloaded from midlet.org. The second one is Soot 2.2.4, a framework for analysing Java bytecode. The last test case is Eclipse SDK 3.2.2

All methods have been successfully analysed, pruned and checked except those that contained sub-routines, referred to unavailable libraries, contained dead code (because the pruning algorithm does not apply if the least fix-points

is $\bot_s$ for some program points) or referred to classes that existed in different versions in the same program. This last case happens in Eclipse and we built conservatively the complete hierarchy of the distributed classes, by taking the union[8]. In all cases, a stack map without conjunction could be obtained from the atoms of the least fix-point, thanks to the heuristic presented in Section 4.2.

The first table shows the main interesting computing times for the three case

	jar size	analysis + pruning	analysis	pruning	checking
midlets	11M	5m54	23%	77%	0m23
Soot	4.4M	3m40	17%	83%	0m11
Eclipse	96M	24m43	26%	74%	1m55

studies. The first column gives the size of the benchmarks (jar files). The second one shows the total time taken by the complete stack map generation procedure (on the producer's side). This time is then divided into the analysis phase (third column) and the pruning phase (fourth column). The last column correspond to the checking time (consumer's side). Clearly, most of the time is spent in pruning, but even this time remains acceptable (three to six times the cost of the analysis), especially since this operation only needs to be performed once, by the code producer. The checking time is short and could be further reduced with a reasonably optimised implementation.

The second table estimates the size of witnesses before and after pruning,

proportion of non-$\top_v$ in	lfp	pruned stack map	ratio
midlets	44%	34%	77%
Soot	67%	42%	63%
Eclipse	58%	39%	66%

in terms of the proportion of pairs p, x for which the value is not $\top_v$. The last column shows the proportion of "positions" (of pairs p, x) that are kept with a non-$\top_v$ value by pruning. We see that the "initial" proportion of non-$\top_v$ is greater in Soot and Eclipse, which indicates that objects (or arrays) are used more often in Eclipse than in midlets (remember that base types are abstracted by $\top_v$). Also, the pruning removes more values in Soot and Eclipse than in midlets (which is not surprising since there are more non-$\top_v$ values to remove, in proportion). In the end, the numbers of non-$\top_v$ in the stack maps are very close for the three test cases.

The first four sub-columns of the last table give the space saved by pruning, both for the class files and the jar files (compressed archives). The numbers correspond to the difference in size with respect to the same file format without stack map. For example, the total jar size for Eclipse with pruned stack maps included is 3.0% greater than the original jar files (without stack maps). In the two columns for the fix-point, since only conjunction-free witnesses can be encoded in class files, we did not include any stack map for the methods whose

[8] Some classes even exist with different super-classes. In this case we just choose one, which is definitely not safe.

witness	fix-point		pruned witness			
representation	extensive		extensive		sparse	
format	.class	.jar	.class	.jar	.class	.jar
midlets	19.8%	7.3%	17.4%	6.8%	16.0%	7.0%
Soot	14.0%	7.2%	11.5%	6.5%	11.5%	7.3%
Eclipse	11.8%	3.3%	9.5%	3.0%	9.6%	3.2%

least fix-point had conjunctions (which is actually quite rare). Note that we can only underestimate the benefit of pruning by doing this. In the case of midlets, for example, the size of the stack maps is reduced from 19.8% to 17.4% of the total initial class files, or from 7.3% to 6.8% of the initial jar files. Therefore there is no significant improvement here since the size of what is shipped (*i.e.*, the jar files with stack maps) is only reduced by less that one percent.

The last two sub-columns of the figure show the effect of a sparse representation of the stack maps obtained after pruning. We see that a sparse representation has little impact on the size, and that the small savings that we get for (some) class files are canceled by the compression phase, and tend to yeld larger jar files (even if the eight-byte alignment of the class files is kept).

We have not tried to encode our stack maps with the new StackMapTable attribute defined by JSR202, which was designed to factorize most of the information. The results would probably be quite different since this format relies on the assumption that the type of variables do not change too often, while the pruning may for example set any variable to $\top_v$ even if it was not modified, as soon as the type information for this variable is not needed anymore.

6 Related work

The formalisation of Java byte code verification has received a lot of attention. Freund and Mitchell [7] prove the soundness of a type system for a very large subset of the Java bytecode with respect to a small-step operational semantics (with explicit stacks). Their model of states is close to ours, but instrumented by tags that keep track of the type of every value. They do not address the problem of inferring types in presence of interfaces. A survey of bytecode verification techniques and solutions to various known difficulties (interfaces, object initialisation, sub-routines) can be found in [11].

The concept of lightweight verification, which is now used in J2ME, was introduced by Rose [16]. Several algorithms were given, with enhancements that allow to reduce the number of program points for which a stack map is necessary, more than what is done in Sun's lightweight bytecode verifier. The issue of verifying interfaces was not considered.

Using sets of types to verify interfaces has been explored by Knoblock and Rehof [10] who analyse an SSA form of the Java bytecode in the Dedekind-MacNeille completion of the type hierarchy. They show that this minimal completion achieves an optimal precision, *i.e.*, every program typable in the power set completion is typable in the Dedekind-MacNeille completion. The analysis

presented in section 3 is therefore very similar to their work. Our representation of the domain differs, though: we use conjunctions of types rather that disjunctions (in both cases, upward/downward-closed sets are not represented in extension). The lattice that we use to abstract values is close to the ideal completion of the type hierarchy (it is a super-set of the ideal completion because the latter further requires that conjunctions be "not empty", in the sense that they must have a lower bound in the hierarchy). Furthermore, our analysis only uses the subset of $\mathit{value}^\sharp$ that is obtained by taking upper bounds of atomic types, which is isomorphic to the Dedekind-MacNeille completion of $\mathit{type} \setminus \{\mathrm{int}\}$. See [6] for an account on completion techniques for posets. Knoblock and Rehof do not prove the correctness of their analysis with respect to a concrete semantics and safety property. Qian [15] proposes a type system for Java bytecode that uses arbitrary disjunctions of reference types to allow the static verification of interfaces. Several safety properties are proved for typable programs (type preservation, possible uses of uninitialized objects, of sub-routines return addresses). The actual inference of types is not detailed. Push [14] has formalised a variant of Qian's bytecode verifier in HOL and proved its correctness with respect to a small-step operational semantics. Again, concrete values are tagged with their type. Goldberg [8] focuses on dynamic loading of classes and proposes a framework for verifying Java class files out of order, while ensuring the global soundness of typing. Class files are verified by a data-flow analysis that uses disjunctions of types (which solves the problem of not knowing the type hierarchy) and yields the minimal set of ordering constraints between types under which the class is type-safe. These constraints are added to a global typing context that is transmitted across invocations of the verifier, and the global safety is defined as the consistency of this context.

We have previously proposed a pruning algorithm for getting weaker abstract interpretation witnesses [3]. Such pruning algorithms were independently studied by Seo, Yang, Yi and Han [17]. The problem that we consider here is different: the goal is not to get a maximal witness in a given lattice, but to get a witness without conjunction, a property that is not monotone. Therefore, directly applying one of the algorithms from [3] would not necessarily help in getting such a witness. The backward computation that we proposed in the same work for distributive analyses (which is the case of the bytecode verification) does not apply either, as shown in the introduction: the backward algorithm performs greatest lower bound operations, which in the present setting introduce conjunctions.

7 Conclusion

We have shown how the notion of pruning provides a viable means of integrating the verification of interfaces into lightweight bytecode verification. This is achieved by combining an extended bytecode analyser and an algorithm for removing conjunctions from the result of the analysis which, together, allows to compute stack maps where interfaces are treated on a par with other types.

The bytecode analysis that we have proposed here adds sets of types to the abstract domain in order to verify interfaces. The ensuing pruning step optimises the typing found by the analyser, reducing all such sets to a singleton, and removing as many typing information as possible while still ensuring the memory safety, *i.e.*, that all memory accesses will be to existing fields of objects. The resulting stack maps can be checked without any overhead compared to existing lightweight bytecode verification and will ensure statically the safety of interface method calls. We also show that it is possible to simplify several aspects of the BCV when constructing an abstract domain that is specific to the memory safety property. In particular, there is no need to distinguish between base types and it is even possible to identify these base type with the $\top_v$ element of the domain (which allows a program to use an address as an integer).

In terms of semantic correctness, we have shown that it is possible to reason directly with an untyped concrete semantics rather than a defensive virtual machine. Both techniques are equally sound, but the latter requires an additional abstraction step that explains the link between the raw state model that we use and the tagged memory objects used in the instrumented semantics. In other words, we use a notion of state that is closer to the actual implementation and, hence, more convincing. In order to complete the picture, the semantics with big-step calls that we used should be related to a small-step semantics with a call stack, but we leave this for further work.

In terms of experiments, we have shown that the technique works well in practice, as we could successfully analyse a large set of Java class files. Furthermore, the idea is not relevant just for Java, but should apply to other object oriented languages with multiple inheritance, since it only relies on the transformation of the poset representing a type hierarchy into a lattice. The results show that it is feasible to compute efficiently conjunction-free stack maps with interfaces ; however, they are disappointing in terms of reducing the stack map size: even though a significant number of variables are set to $\top_v$ by pruning, this is not enough for a sparse coding to be more efficient than a naive coding of stack maps, especially as class files are eventually compressed.

As we said before, in this study we considered one aspect of the security of Java bytecode, *viz.*, the memory safety. Further work should extend the formalisation proposed here to prove that for example access control properties are also ensured by the verifier. In another direction, our stack map generator should be extended to produce stack maps in the StacMapTable format proposed for Java.

References

1. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *Proc. of the 3rd Int. Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, volume 2294 of *LNCS*, pages 32–45. Springer, 2002.
2. F. Besson, T. Jensen, and T. Turpin. Computing stack maps with interfaces. Technical Report 1879, Irisa, 2007.

3. F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation-based proofs. In *Proc. of the 16th European Symp. on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 268–283. Springer, 2007.
4. G. Bracha, T. Lindholm, W. Tao, and F. Yellin. *CLDC Byte Code Typechecker Specification*. Sun Microsystems, 2003.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.
6. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 1990.
7. S. N. Freund and J. C. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, 2003.
8. A. Goldberg. A specification of java loading and bytecode verification. In *Proc. of the 5th ACM conference on Computer and Communications Security (CCS'98)*, pages 49–58. ACM, 1998.
9. JSR 202 Expert Group. *Java Class File Specification Update*, 2006. Sun Microsystems.
10. T. B. Knoblock and J. Rehof. Type elaboration and subtype completion for java bytecode. *ACM Transactions on Programming Languages and Systems*, 23(2):243–272, 2001.
11. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
12. X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. of the 25th ACM Symp. on Principles of Programming Languages (POPL'98)*, pages 391–403. ACM, 1998.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd edition)*. Prentice Hall, 1999.
14. C. Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *Proc. of the 5th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 89–103. Springer-Verlag, 1999.
15. Z. Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. *Formal Syntax and Semantics of Java*, pages 271–312, 1999.
16. E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3-4):303–334, 2003.
17. S. Seo, H. Yang, K. Yi, and T. Han. Goal-directed weakening of abstract interpretation results. *ACM Transactions on Programming Languages and Systems*, 29(6):39–, 2007.