An Abstract Interpretation Approach for Enhancing the Java Bytecode Verifier

Roberto Barbuti\(^1\), Nicoletta De Francesco\(^2\) and Luca Tesei\(^3\)

\(^1\)Dipartimento di Informatica Università di Pisa Largo B. Pontecorvo, 1 - 56127 Pisa - Italy
\(^2\)Dipartimento di Ingegneria dell’Informazione - Università di Pisa - Via Diotisalvi, 2 - 56126 Pisa - Italy
\(^3\)Dipartimento di Matematica e Informatica - Università di Camerino - Via Madonna delle carceri, 9 - 62032 Camerino (MC) - Italy

Email: luca.tesei@unicam.it

The Java Virtual Machine embodies a verifier that performs a set of checks on Java bytecode programs before their execution. The verifier carries out an efficient data-flow analysis applied to a type-level abstract interpretation of the code. The implementations of the bytecode verifier presented a significant problem with programs compiled with the Sun Java compiler (until version 1.4.1): there were legal Java programs which were correctly compiled into bytecode that was rejected by the verifier. The problem was fixed by removing, in version 1.4.2 and following, some interesting features in the compilation of the try-finally Java construct. Because removing such features has a cost in terms of memory space, in this paper we propose to enhance the bytecode verifier to accept such programs, maintaining the space efficiency of the previous versions of the compiler. We define an abstract interpretation framework in which we model the enhanced version of the verifier. The defined abstract interpretation framework can be considered a good basis for other static analyses of bytecode programs.

Received 00 Month 2004; revised 00 Month 2004

1. INTRODUCTION

Java programs are compiled into an intermediate language which is executed by the Java Virtual Machine (JVM in the following) [1]. The intermediate language, i.e. the instruction set of the JVM, is usually called Java bytecode or, in several papers in the literature, JVML. Throughout the paper we refer to this language as JVML or simply bytecode.

Since JVML programs can be loaded from the network, security problems may arise. For this reason, the JVM embodies a bytecode verifier that performs a set of checks on JVML programs before their execution. The aim of these checks is to prevent the execution of malicious or wrong code that could corrupt the integrity of the host. In particular, the verifier performs a data-flow analysis applied to a type-level abstract interpretation of the JVM. We refer to this verifier as the standard one and to its verification as the standard verification.

Given the importance of bytecode verification, a lot of research efforts have been dedicated both to its formalisation [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] and to study extensions able to accept larger classes of correct programs than the standard verifier does [14, 15, 16, 10, 8].

Among others [17], the bytecode verifier presented a significant problem when used on programs compiled with the Sun Java compiler, until Version 1.4.1: there were legal Java programs that were correctly compiled into bytecode programs that were rejected by the verifier. This problem has been pointed out in [18, 19, 20], where examples of these programs are reported. The source of the problem is the mechanism of subroutines that were used to compile the Java try-finally construct. The verification of bytecode programs is significantly more complex in presence of subroutines since they have multiple calling points, multiple return points and they are executed in the same activation record of the main code [1].

To avoid the problem, from Version 1.4.2, the Sun Java compiler, by default, does not use subroutines for compiling the try-finally construct. Instead, the
compiler will inline subroutine code in places where the subroutines are called. As a consequence, the compiler generates more bytecode than before. Since the class file format limits the size of methods, those methods with large or nested finally blocks may exhaust this limit and fail to compile. Thus, avoiding the original problem generated a new different problem.

Moreover, the Java SE 6 adopts a style of verification which is based on type checking instead of type inference. The idea is to split the old verification process into two phases: type inferencing (off-line) and type checking (on-line). The new verifier (called the type-checking verifier) no longer needs to infer types because it uses type information embedded in the Java class file to check the bytecode type consistency. There are a few consequences of this change: when doing bytecode instrumentation, it is necessary to adjust StackMapTable attributes to reflect changes made to the bytecode. Otherwise the modified class files will probably fail the new verifier [21].

Currently, it is possible to resort to jar and ret instructions and to the old verifier using a non-standard option of the compiler. This possibility is an escape whenever the generated bytecode is rejected because it exceeds the limits of the class file format due to large or nested finally blocks, a scenario that is not unusual in the JavaServer Pages (JSP) programming. This option is also useful when optimisation or control-flow analysis tools are used on bytecode, as inline code may corrupt their work.

All things considered, it can be said that the subroutine problem in bytecode verification remains unresolved and it should be still investigated, with the goal of finding better solutions. The above cited research efforts - including also this work - carried out to treat safely non-structured code as that of subroutines could be exploited to make available a Java compiler and verifier that are free of tricky problems not depending on the program to be compiled and/or executed.

This paper provides two contributions to the study of Java bytecode verification. The first one is to give a formal framework to reason about bytecode verification. The framework is based on abstract interpretation [22, 23, 24]. An abstract interpreter executes the program in an abstract (approximated) way to statically check dynamic properties. The actual domain of computation (called concrete domain) is replaced by an abstract domain and the operators of the concrete computation are replaced by correct abstract operators. We define the verification process as a formal abstract semantics of JVM. The formalisation can be used to compare different models and implementations of the verifier.

The second, and main, contribution of the paper is to propose an enhanced verifier able to certify programs obtained by the compilation of the try-finally by using bytecode subroutines. We formalise the enhanced verifier in the abstract interpretation framework [25]. The enhanced verifier is based on the same algorithm of the standard one. The difference is that it uses a domain of types that is more rich that the one used in standard verification. The idea is that the subroutine problem can be solved simply by using types carrying information, at each program point, both on the types of values stored in registers and whether such registers are assigned on all paths leading to the program point itself. We show how the proposed enhanced verifier correctly accepts the bytecode programs presented in [18, 19, 20].

It is important to notice that an accurate implementation of the proposed extension maintains the efficiency of the standard verifier, in particular for what concerns subroutines. It is also important to remark that the formal definition of the concrete and abstract domains represents a clear basis for the construction of other static analysis tools for bytecode programs. This is shown in [26] where also the standard bytecode verifier is defined in our abstract interpretation framework.

The paper is organised as follows. Section 2 defines a fragment of the JVM, specifies the semantics of bytecode programs and recalls the behaviour of the standard verifier. Section 3 outlines informally our approach. Section 4 defines the concrete domain of computation suitable for the abstract interpretation. Section 5 defines our abstract interpreter showing how it correctly handles subroutines. Section 6 contains some notes on a possible implementation. Section 7 discusses related works and concludes. For the sake of readability, some proofs have been placed in the Appendix.

2. JVM AND JAVA BYTECODE VERIFIER

In this section we define a fragment of the JVM and we specify the semantics of bytecode programs. This semantics is used as a basis to define the concrete domain of the abstract interpretation. We then outline the Java bytecode verifier.

2.1. A model of the Java virtual machine

The result of the compilation of a Java program is a set of class files. A class file is generated by the Java compiler from each class or interface definition of the program. It contains all the details needed by the JVM for creating objects and executing methods of the class. At this level the implementation of each method is a bytecode program resulting from the compilation of the corresponding source code. In general bytecode is a form of intermediate code, a binary representation of an executable program designed to be executed by an interpreter or virtual machine rather than by hardware. In the context of Java compilation, bytecode is constrained to be a stack based assembly language whose instructions operate on a local stack and some local registers that can store different types of basic values. It is executed by the JVM. In [1] the full set
of bytecode instructions and the JVM architecture are described by a technical prose.

Figure 1 shows the restricted set \( \text{Instr} \) of bytecode instructions that we consider in this paper\(^4\). This choice follows the approach of \([2, 5, 27, 8]\) that focus, as this paper does, on the subject of subroutines and bytecode verification and use very similar sub-languages.

It is important to remark, however, that our abstract interpretation framework can be extended (with some technical effort) to the whole set of bytecode instructions, as we outline at the end of Section 6.

Let us start modelling the execution of a single bytecode program by the JVM. This is in accordance with our need of having a basic semantics on which the bytecode verification can be defined. The verification is performed for each method separately.

As we outlined above each method is compiled into a bytecode program. We model it by a finite sequence of bytecode instructions at addresses \(0, 1, 2, \ldots, K - 1\). Moreover, given a bytecode program, we suppose to have:

- a function \( P: \{0, 1, 2, \ldots, K - 1\} \rightarrow \text{Instr} \) such that \( P(i) \) is the instruction at address \( i \)
- \( N \in \mathbb{N} \) as the number of local registers required for the method
- \( \text{max\_stack\_height} \in \mathbb{N} \) as the maximum height that the stack can have during the execution of the method
- \( m \in \mathbb{N} \) (\( m \leq N \)) as the number of the parameters of the method.

The values for \( N \), \( \text{max\_stack\_height} \) and \( m \) are stored in the class file and typically result from the compilation.

The types of JVM\( \text{L} \) include basic types, such as \texttt{int} or \texttt{double}, object references (not considered in the paper), and types of the form \( \text{ret}(L) \) for each subroutine in the bytecode program. A type \( \text{ret}(L) \) represents valid return addresses \( \ell \) from a subroutine starting at program point \( L \), i.e., addresses \( \ell \) of instructions following\(^6\) a \texttt{jsr} \( L \) instruction. Moreover, the domain of types includes \( \Omega \), which represents uninitialised values. We denote by \( \text{Values} \) the semantic domain of possible values. It is composed by pairs of actual values and their corresponding types\(^7\):

\[
\text{Values} = \left( \mathbb{Z} \times \{\text{int}\} \right) \cup \{(\omega, \Omega)\} \cup \{((\ell, \text{ret}(L)) | \ell, L \in \{0, 1, \ldots, K - 1\}, \ell > 0, P(\ell - 1) = \texttt{jsr} L)\}
\]

where \( \mathbb{Z} \) are integer numbers and \( \omega \) represents an uninitialised value. The annotation of actual values with types in the concrete semantics does not correspond to the implementation of the JVM, which stores only values. We use this technical trick to simplify the definition of the abstract interpretation.

The computational state of a program is defined as follows:

**Definition 2.1. (State of a Bytecode Program)**

A state of a bytecode program is a pair \((i, (M, St))\) where:

- \( i \in \{0, 1, \ldots, K - 1\} \) is a program control point, i.e. the address of an instruction
- \( M: \{1, 2, \ldots, N\} \rightarrow \text{Values} \) is a function that gives the local registers value. For each local register \( x \), \( M(x) \) is the pair of the current value of \( x \) and its corresponding type. The set of all such functions is denoted by \( \text{Mem}(`\text{Values}) \)
- \( St \) is a stack representing the local operand stack at instruction \( i \). It also holds pairs of values and types and the usual operations of push and pop are defined on it. We represent stacks as sequences of values separated by dots in which the leftmost value is the top. We use \( (\cdot) \) to represent an empty stack and \(|St|\) to denote the length (or height) of the stack \( St \). We denote by \( \text{Stack}(`\text{Values}) \) the set of all stacks whose height is at most the constant \( \text{max\_stack\_height} \):

\[
\text{Stack}(`\text{Values}) = \left\{ () \right\} \cup \{s_1 \cdots s_{i-1}, s_i | \forall i, 1 \leq i \leq p, s_i \in \text{Values}, 1 \leq p \leq \text{max\_stack\_height}\}
\]

The set of all possible bytecode program states is

\[
\text{States} = \{0, 1, \ldots, K - 1\} \times \text{Mem}(`\text{Values}) \times \text{Stack}(`\text{Values})
\]

**Definition 2.2. (Initial States)**

A state of a bytecode program is initial if it is of the form \((0, (M_0, ())\) where \( M_0 \) is such that:

- for all \( x \in \{1, 2, \ldots, m\} \), \( M_0(x) \) is the actual value of the \( x \)-th parameter of the method (that is supposed to be given by the caller)
- for all \( x \in \{m + 1, m + 2, \ldots, N\} \), \( M_0(x) = (\omega, \Omega) \)

\(^4\) For the sake of simplicity we use a slightly different version of the mnemonic codes used in [1].

\(^5\) For simplicity we use sequential integers to represent them. Actually, bytecode instructions are of variable length, thus in a real implementation the addresses are, in general, different.

\(^6\) Bytecode programs whose last instruction is a \texttt{jsr} \( L \) are considered malformed since the return address from this call of subroutine would be \( K \), which is not in the boundary of the program. Such programs are immediately rejected without any verification.

\(^7\) The indicated values are a subset of the actual values handled by the JVM. We use only the values that can be generated in bytecode programs composed by instructions in the restricted set.
The rules of Figure 2 define a relation $\rightarrow_{\text{next}}$ between states of a bytecode program. This relation represents the actions that the JVM does starting from the initial state and executing a given bytecode program until a final state. We use $M[x/v]$ to denote the memory $M$ updated on the entry $x$ with value $v$. The rules are straightforward. We remark that the rule for astore only manages return addresses (from subroutines) and not also object references (as in the full JVM) because in our restricted language we do not have object features. We also remark that the next state of a final state is the same state (ireturn rule). This is useful for the definition of the concrete domain of the abstract interpretation.

It is important to note that in the rules the actual types of the operands are checked with respect to the ones required by the instructions and that the control of stack overflow and underflow is performed. This corresponds to the so-called defensive JVM [28]. The real implementation does not perform these controls because it trusts the verification carried out before the execution. We prove in Theorem 5.1 that the verification is correct with respect to this defensive JVM, i.e. if a bytecode program passes the verification then the defensive JVM would execute it without runtime errors. Thus, the program can be safely executed by a non-defensive JVM.

2.2. The bytecode verifier

A bytecode program, in order to be properly executed, must respect a set of conditions [1], among which type correctness (the arguments of an instruction belong to the types expected by the instruction), no operand stack underflow and overflow, code containment (destinations of jumps are addresses in the code), register initialisation (registers must be initialised before being used). It would be expensive to check these conditions at run time, thus the purpose of the bytecode verifier is to check them statically before execution, as described in [1]:

“...the verifier checks the code array of the Code attribute for each method of the class file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the following is true:

- The operand stack is always the same size and contains the same types of values.
- No local variable is accessed unless it is known to contain a value of an appropriate type. [...]
- All opcodes have appropriate type arguments on the operand stack and in the local variable array.

..."

Almost all existing bytecode verifiers implement the algorithm described in [1]. It is a data-flow analysis applied to a type-level abstract execution of the program: instead of operating on data, the instructions are simulated on types. For example, an iload $x$ instruction checks that register $x$ contains the type int and in this case pushes int onto the stack. If the contents of $x$ is different from int, then an error is found. The algorithm considers a partially ordered set of types, containing a type $\top$ representing a value that has not been initialised.

The verifier maintains a table $S$ having, for each instruction address $i$, a row $S_i = (M_i, St_i)$ representing the abstract state (containing types instead of actual values) before the execution of $i$. Instruction $i$ is executed in state $S_i$ and the state produced (the after-state of $i$) is merged with the state $S_j$ of each successor instruction $j$, producing a new state for $S_j$. Merging is necessary since, due to the conditional and unconditional jumps and the calls (returns) to (from) subroutines, there are instructions corresponding to a join of different paths of the control flow graph. Merging two states consists in merging the types of each memory register and stack element. Only stacks with the same length can be merged. Merging different types gives as result $\top$. Hence this type represents also a contradictory type, resulting from the merge of incompatible types. The verifier continues to iteratively execute the program instructions until an error is encountered or a fixpoint is reached. In the second case, the bytecode is certified and accepted for execution.
Figure 3 shows the results of compilation and verification of a Java method. The second column in the figure shows the result of the compilation of the method in the first one. The other columns show the type assignments, produced by the verifier, to the registers and to the stack.

Registers 1 and 2 contain the types of parameters b and x, respectively. At the instruction labelled by A, the state after the instruction ifeq A is merged with the state after the instruction istore 2. Thus the type of register 3, obtained by merging the types int and T, becomes T.

The verification of subroutines complicates significantly bytecode verification [1]. First, since the bytecode is not structured, the definition of the boundaries of subroutines is not simple. Often a set of assumptions and constraints is made to simplify this computation (see [10] and [27] for a discussion). Secondly, the first instruction of a subroutine is a merge point of the control flow graph, causing the registers and the stack to be merged at that point, and this can lead to an excessive loss of precision. If a register is not used by the subroutine it can safely maintain, at the return point, the type held before the call, which may be different for different calls. To (partially) avoid this loss of precision, the verifier handles unused registers in a special way [9, 10]. The state at the first instruction of a subroutine starting at a program point L is obtained by merging the states after all the instructions occurring in the control flow graph, causing the registers and the stack to be merged.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iconst</td>
<td>( P(i) = \text{iconst } c, )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (i + 1, (M, (c, \text{int}).St)) )</td>
</tr>
<tr>
<td>iload</td>
<td>( P(i) = \text{iload } x, )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (i + 1, (M, (c, \text{int}).St)) )</td>
</tr>
<tr>
<td>istore</td>
<td>( P(i) = \text{istore } x, \ St = (c, \text{int}).St', \ i + 1 \in {0, 1, \ldots, K - 1} )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (i + 1, (M[[c, \text{int}]/x], St')) )</td>
</tr>
<tr>
<td>astore</td>
<td>( P(i) = \text{astore } x, \ St = (\ell, \text{ret}(L)).St', \ i + 1, \ell, L \in {0, 1, \ldots, K - 1} )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (i + 1, (M[[\ell, \text{ret}(L)]]/x], St')) )</td>
</tr>
<tr>
<td>ifeq 0</td>
<td>( P(i) = \text{ifeq } L, \ St = (0, \text{int}).St', \ L \in {0, 1, \ldots, K - 1} )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (L, (M, St')) )</td>
</tr>
<tr>
<td>goto</td>
<td>( P(i) = \text{goto } L, \ L \in {0, 1, \ldots, K - 1} )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (L, (M, St)) )</td>
</tr>
<tr>
<td>jsr</td>
<td>( P(i) = \text{jsr } L, )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (L, (M, (i + 1, \text{ret}(L))).St)) )</td>
</tr>
<tr>
<td>ret</td>
<td>( P(i) = \text{ret } x, \ x \in {1, 2, \ldots, N}, \ M(x) = (\ell, \text{ret}(L)) )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (\ell, (M, St)) )</td>
</tr>
<tr>
<td>ireturn</td>
<td>( P(i) = \text{ireturn}, \ St = (c, \text{int}).St' )</td>
</tr>
<tr>
<td></td>
<td>( (i, (M, St)) \rightarrow_{\text{next}} (i, (M, St)) )</td>
</tr>
</tbody>
</table>

**FIGURE 2.** Computational steps of the defensive JVM
FIGURE 3. An example of compilation and verification of a Java method

<table>
<thead>
<tr>
<th>int m0(boolean b, int x) {</th>
<th>Memory</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int z;</td>
<td>load 1</td>
<td>(int)</td>
</tr>
<tr>
<td>if (b)</td>
<td>ifeq A</td>
<td>(int)</td>
</tr>
<tr>
<td>z=x; x=1;</td>
<td>iload 2</td>
<td>()</td>
</tr>
<tr>
<td>return 0;</td>
<td>istore 3</td>
<td>(int)</td>
</tr>
<tr>
<td>}</td>
<td>iconst 1</td>
<td>()</td>
</tr>
<tr>
<td>A:</td>
<td>istore 2</td>
<td>(int)</td>
</tr>
<tr>
<td>return</td>
<td>iconst 0</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>(int)</td>
</tr>
</tbody>
</table>

the registers are handled in a different way depending on whether they are modified or not inside the subroutine code. Consider an occurrence of \texttt{jsr} \texttt{L} at address \(i\) and its successor instruction at \(i + 1\). Consider the state \(S_i = (M_i, St_i)\) at \(i\) and the state \(S_{\text{ret}} = (M_{\text{ret}}, St_{\text{ret}})\) after the execution of the instruction \texttt{ret} \(x\) returning from the subroutine. The state at the program point \(i + 1\) is computed as follows. The operand stack \(St_{i+1}\) is obtained by merging the previous \(St_{i+1}\) with \(St_{\text{ret}}\). For each register \(x\) modified by the subroutine, the new \(M_{i+1}(x)\) is obtained by merging the old \(M_{i+1}(x)\) with \(M_{\text{ret}}(x)\), following the usual behaviour of the verifier. Instead, for each register \(x\) not modified by the subroutine, the new \(M_{i+1}(x)\) is obtained by merging the old \(M_{i+1}(x)\) with \(M_{\text{ret}}(x)\), that is the type of \(x\) before the call. Hence a register not used inside the subroutine can have different contents after different calls.

Figure 4, in the second column, shows the result of the verification of a Java bytecode program which is rejected by the standard verifier, since it fails to type the instructions \([18, 19]\). The figure also shows the typing assignment to the instructions produced by the verifier, and the registers modified by the subroutine. The subroutine, starting at the address \(L\), is called from two \texttt{jsr} instructions: at the first \texttt{jsr} register 2 has type \(\top\) and register 3 has type \(\texttt{int}\), while, at the second call, register 2 has type \(\texttt{int}\) and register 3 has type \(\top\). Thus, the body of the subroutine is abstractly executed starting from a memory which is the least upper bound of the two calling points, that is \(M_L(1) = \texttt{int}, M_L(2) = M_L(3) = M_L(4) = \top\). Note that there is a path from \(L\) to the last instruction of the subroutine (\(B\): \texttt{ret 4}\) where register 2 is not modified and another path from \(L\) to \(B\) in which 2 is assigned an integer value. Hence, at the instruction \(B\) the paths are merged and register 2 holds the type \(\top\), which is the result of merging \(\texttt{int}\) and \(\top\). Since 2 belongs to the set of registers modified by the subroutine, it is assigned \(\top\) in the memory for the instructions following the two calls. Thus it holds \(\top\) both at the instruction \texttt{goto C} and at the address \(C\), where an error is signalled, since \(\top\) cannot be loaded. Since register 3 is not modified by the subroutine, the type for it produced by the subroutine \((\top)\) is not considered, and for each call the type before the call is propagated to the return point. Thus, it has type \(\texttt{int}\) at the first return point (where it is used) and \(\top\) at the second.

3. OUTLINE OF OUR APPROACH

The aim of this paper is twofold: a) to give an abstract interpretation framework to reason about the analysis of bytecode programs and b) to formalise in this framework an enhanced verifier able to certify a larger class of programs with respect to the standard one.

3.1. Abstract interpretation for the analysis of bytecode programs

We give a basis to reason about bytecode programs by means of abstract interpretation and in this setting we define the verifier as a formal abstract semantics of JVML. The formalisation gives a precise framework which points out the main features of verification and allows to compare different models and implementations of the verifier. Following the abstract interpretation approach we first define a concrete accumulating semantics which associates each instruction address with the set of all possible states in which the JVM can be when the program counter contains this address.

The abstraction is performed by substituting abstract values for sets of concrete ones: for example, a set of values with the same type is abstracted into the type itself. The abstraction of values induces a natural abstraction on memories, stack configurations and states. The abstract semantics operators are then defined on these abstract domains. The construction is proved to respect the correctness conditions of abstract interpretation. We remark that the abstraction we define is constrained: in order to obtain a significant abstraction the concrete states must be consistent with the structure of programs.

A main point of the approach is that the abstract interpreter is parametric with respect to the domain of types: different verifiers can be defined based on the same abstract rules (the verification algorithm), but using different type domains. This allows us to compare the precision of different verifiers: the higher the level of precision is, the wider is the class of type-safe bytecode programs accepted by the corresponding
We express the enhanced verifier using this framework and we show that it is able to certify a larger class of programs with respect to the standard verifier.

3.2. The enhanced verifier

Reconsider the example in Figure 4. While the bytecode is rejected by the standard verifier, the Java source program (method m1 in the figure) is correctly accepted by the Sun Java compiler. In fact, in all executions an integer will be held by register 2 at the C address: either the register holds the same value held before the subroutine call or a new integer value has been assigned to it by the subroutine (if the specific branch was chosen). Instead, the verifier assigns to it the inconsistent type.

The problem is that the standard verifier does not distinguish between a typing incompatible with int at all (e.g. an address) and a typing compatible with it. The domain of types seems to be too poor: the type $\tau$ represents three situations that could be handled in different ways: 1) the undefined value, i.e. the type of an uninitialised (not assigned, untouched) register; 2) an error, when $\tau$ derives from the merging of two incompatible types; 3) a type of a register (stack elements) which, at a merge point, has been assigned a different type for each different path.

We propose a solution where the domain of types is extended to contain a different type for each different situation among those listed above. The new domain of types contains the type unt $\tau$ to represent the undefined value, i.e. the type of an uninitialised register, the top value $\top$ which represents a contradictory type (resulting from the merge of incompatible types), and, for each type $\tau$, the type unt $\tau$ which means that the register or stack element (at a merge point) has been assigned with a value of type $\tau$ on some program paths and has not been initialised on the other paths. In terms of values, the type unt $\tau$ represents all the values represented by $\tau$ plus the uninitialised value.

The verification of the method of Figure 3, using the new type domain, is given in Figure 5, where the type of register 3, at the instruction A, is now unt $\tau$ instead of $\tau$. This corresponds to the intuition that register 3 on a path to A receives an integer value, and on another path is untouched. The verification succeeds (in both verifiers) because the value of register 3 is not used.

Using this domain of types, we define in a different way the type to be assigned at the return points to the registers modified by the subroutine (the other registers are handled as in standard verification). Consider a call at instruction i and a modified register x in the subroutine and let $S_{ret} = (M_{ret}, S_{ret})$ be the state after the execution of the instruction ret returning from the subroutine. When we compute $M_{ret+1}(x)$, instead of using $M_{ret}(x)$ as in standard verification, we use a function that combines $M_i(x)$ and $M_{ret}(x)$: the main point is that a type $M_i(x) = \tau$ held by a register when a subroutine is called, when combined with the type $M_{ret}(x) = \text{unt } \tau$, computed by the subroutine, gives $\tau$ as result. In this way, the types held by registers before a subroutine call are used for deducing the types, at the return point, of both unmodified and modified registers.

Figure 6 shows the result of the verification of the program in Figure 4 (first column) performed with the new domain. We have that the value of register 2 at the beginning of the subroutine (instruction L) is unt $\tau$, since, when L is reached from the first call, the value of register 2 is uninitialised (unt), while when L is reached
Consider the complete lattice

$$\langle p(\text{States}), \subseteq, \cup, \cap, \{\}, \text{States} \rangle$$

where $p$ is the powerset operator, the set union is the least upper bound (lub) operator, the set intersection is the greatest lower bound (glb) operator, $\{\}$ is the bottom element and $\text{States}$ is the top element. This notation is used all over the paper to denote complete lattices and their components (recognisable by the order in which they appear in the tuple). A lattice could be denoted only by its set and its order relation when the other components are clear from the context or when they are useless in the argumentation.

**Proposition 1.** The operator $\text{next}^C$ is monotone in $\langle p(\text{States}), \subseteq \rangle$:

$$\forall S, S' \in p(\text{States}), S \subseteq S' \Rightarrow \text{next}^C(S) \subseteq \text{next}^C(S')$$

**Proof.** The thesis follows directly from the definition of $\text{next}^C$. \hfill $\Box$

**Definition 4.2.** (Execution Set) Given a bytecode program and an initial state $s_0 = (0, (M_0, ()))$, the Execution Set $C_{s_0}$ is the least upper bound, in $\langle p(\text{States}), \subseteq \rangle$, of the increasing chain defined, for all $n \in \mathbb{N}$, as follow:

$$C_{s_0}^0 = \{(0, (M_0, ()))\}$$

$$C_{s_0}^{n+1} = C_{s_0}^n \cup \text{next}^C(C_{s_0}^n)$$

The set $C_{s_0}$ represents one execution of the program, the one starting from the initial values in $s_0$.

**Definition 4.3.** (Accumulating Semantics) Given a bytecode program, the set of all possible executions is denoted $C$ and it is defined as follows:

$$C = \{(i, (M, St)) \in C_{s_0} \mid s_0 \text{ is an initial state}\}$$

The set $C$ is infinite most of the time. This makes the checking of runtime properties undecidable. We define an abstract interpreter that approximates the real computations of $C$ providing a finite abstract representation of it in which the information of actually computed values is lost and only information on the

<table>
<thead>
<tr>
<th>int m0(boolean b, int x) {</th>
<th>Memory</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>iload 1</td>
<td>{1:int, 2:int, 3:unt}</td>
</tr>
<tr>
<td>if(b)</td>
<td>iload 2</td>
<td>{1:int, 2:int, 3:unt}</td>
</tr>
<tr>
<td>z=x; x=1;</td>
<td>istore 3</td>
<td>{1:int, 2:int, 3:unt}</td>
</tr>
<tr>
<td>return 0;</td>
<td>icall 1</td>
<td>{1:int, 2:int, 3:int}</td>
</tr>
<tr>
<td>}</td>
<td>istore 2</td>
<td>{1:int, 2:int, 3:int}</td>
</tr>
<tr>
<td>A:</td>
<td>istore 0</td>
<td>{1:int, 2:int, 3:unt}</td>
</tr>
<tr>
<td>ireturn</td>
<td></td>
<td>{1:int, 2:int, 3:unt}</td>
</tr>
</tbody>
</table>

FIGURE 5. An example of verification of a Java method with the enhanced type domain

from the second call the value is an integer. At the ret instruction the contents of register 2 is unt, representing the fact that, in the subroutine, on one path register 2 is not touched and in the other path it is assigned an integer value. When the state of the return point after the first jar is calculated, the combination of the type held by register 2 before the call (unt) with the return type (unt, int) is unt. Instead, for the second call, the combination of the type before the call (int) with the return type (unt, int) is int. Intuitively, if register 2 contains int before the subroutine call and the paths of the subroutine either assign int to it or leave it untouched, at the return point register 2 must contain int. This assignment eliminates the error that the standard verification produces at instruction iload at program point $C$. Hence, the type int for register 2 is propagated until the last instruction of the program, which thus results type-safe. Note that, if $\tau \neq \tau'$, the combination of a type $\tau$ with unt,$\tau'$ would result in an incorrect type and the same for the combination of unt,$\tau$ and unt,$\tau'$.

4. AN ACUMULATING SEMANTICS OF THE JVML FRAGMENT

The definition of the concrete semantics domain follows the classical approach introduced in [22]. In the abstract interpretation framework this domain is generally referred as collecting semantics or accumulating semantics or static semantics [29]. The accumulating semantics of a bytecode program either associate with every instruction $i$ the set of all JVM states, that is to say $(M, St)$ pairs, that can occur when the control reaches instruction $i$.

**Definition 4.1.** (One-step Operator) Given a set $S$ of bytecode program states, the application of the One-step Operator $\text{next}^C$ yields the set of bytecode program states that can be reached in one step of computation, starting from the states in $S$. Formally,

$$\text{next}^C(S) = \{(j, (M', St')) \mid (i, (M, St)) \in S, (i \rightarrow \text{next}^C (j, (M', St'))\}$$

where $\text{next}^S$ is the greatest lower bound (glb) operator, the set intersection is the least upper bound (lub) operator, the set union is the powerset operator, $\{\}$ is the bottom element and $\text{States}$ is the top element. This notation is used all over the paper to denote complete lattices and their components (recognisable by the order in which they appear in the tuple). A lattice could be denoted only by its set and its order relation when the other components are clear from the context or when they are useless in the argumentation.

**Proposition 1.** The operator $\text{next}^C$ is monotone in $\langle p(\text{States}), \subseteq \rangle$:

$$\forall S, S' \in p(\text{States}), S \subseteq S' \Rightarrow \text{next}^C(S) \subseteq \text{next}^C(S')$$

**Proof.** The thesis follows directly from the definition of $\text{next}^C$. \hfill $\Box$

**Definition 4.2.** (Execution Set) Given a bytecode program and an initial state $s_0 = (0, (M_0, ()))$, the Execution Set $C_{s_0}$ is the least upper bound, in $\langle p(\text{States}), \subseteq \rangle$, of the increasing chain defined, for all $n \in \mathbb{N}$, as follow:

$$C_{s_0}^0 = \{(0, (M_0, ()))\}$$

$$C_{s_0}^{n+1} = C_{s_0}^n \cup \text{next}^C(C_{s_0}^n)$$

The set $C_{s_0}$ represents one execution of the program, the one starting from the initial values in $s_0$.

**Definition 4.3.** (Accumulating Semantics) Given a bytecode program, the set of all possible executions is denoted $C$ and it is defined as follows:

$$C = \{(i, (M, St)) \in C_{s_0} \mid s_0 \text{ is an initial state}\}$$

The set $C$ is infinite most of the time. This makes the checking of runtime properties undecidable. We define an abstract interpreter that approximates the real computations of $C$ providing a finite abstract representation of it in which the information of actually computed values is lost and only information on the
### FIGURE 6. An example of verification of try-finally with the enhanced type domain

<table>
<thead>
<tr>
<th>Memory</th>
<th>Stack</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{int} m1(boolean b) {</td>
<td>\text{iload 1}</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>\text{int} i;</td>
<td>\text{ifeq} A</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>try {</td>
<td>\text{iconst 1}</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>if (b)</td>
<td>\text{istore} 3</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>return 1;</td>
<td>\text{jsr} L</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>i = 2;</td>
<td>\text{iload} 3</td>
<td>{1:int, 2:unt_int, 3:int, 4:}\top</td>
</tr>
<tr>
<td>} finally {</td>
<td>\text{ireturn}</td>
<td>{1:int, 2:unt_int, 3:int, 4:}\top</td>
</tr>
<tr>
<td>if (b) A:</td>
<td>\text{iconst} 2</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>i = 3;</td>
<td>\text{istore} 2</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>}</td>
<td>\text{jsr} L</td>
<td>{1:int, 2:unt, 3:unt, 4:unt}</td>
</tr>
<tr>
<td>return i;</td>
<td></td>
<td>{1:int, 2:unt, 3:unt, 4:}\top</td>
</tr>
<tr>
<td>L:</td>
<td>\text{astore} 4</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:unt}</td>
</tr>
<tr>
<td>\text{iload} 1</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}</td>
<td>()</td>
</tr>
<tr>
<td>\text{ifeq} B</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}</td>
<td>(int)</td>
</tr>
<tr>
<td>\text{iconst} 3</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}</td>
<td>()</td>
</tr>
<tr>
<td>\text{istore} 2</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}</td>
<td>(int)</td>
</tr>
<tr>
<td>B: ret 4</td>
<td>{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}</td>
<td>()</td>
</tr>
<tr>
<td>C: iload 2</td>
<td>{1:int, 2:unt, 3:unt, 4:}\top</td>
<td>()</td>
</tr>
<tr>
<td>\text{ireturn}</td>
<td>{1:int, 2:unt, 3:unt, 4:}\top</td>
<td>()</td>
</tr>
</tbody>
</table>

Types is present. The abstract execution always terminates and makes the checking of the above runtime properties decidable. The following sections concentrate on the construction of such abstract interpretation.

The concrete domain of the abstract interpretation is defined, as usual, as a powerset. The natural candidate is the following:

**Definition 4.4. (Concrete Domain)** The concrete domain of the abstract interpretation is the complete lattice \(\wp(\text{States}), \subseteq, \sqcup, \sqcap, \{\}, \text{States}\).

### 5. The Structure of the Abstract Domain

In this section we show how to construct incrementally an abstract domain for the concrete domain defined in the previous section starting from a given domain of types. We also show how to construct the Galois insertions connecting the concrete domain and the constructed abstract domains. These definitions will be used to define the enhanced verifier in the following sections. Note that they also represent a good formal and general basis for other improvements of the standard verifier or for different static analyses of the bytecode.

The construction is parametric with respect to a given finite\(^8\) complete lattice \((A, \sqsubseteq_A, \sqcup_A, \sqcap_A, \top_A, \bot_A)\), whose elements are intended to represent sets in \(\wp(\text{Values})\), together with a given Galois insertion \([22, 24]\) between them. Let us recall the general definition:

\(^8\)We use finite lattices. This implies also that they are complete. It is possible, however, to use the same construction starting from an infinite complete lattice.

**Definition 5.1. (Galois Connection/Insertion)** Let \((C, \subseteq)\) and \((A, \sqsubseteq)\) be two complete lattices. Two functions \(\alpha : C \rightarrow A\) and \(\gamma : A \rightarrow C\) form a Galois connection between \((C, \sqsubseteq)\) and \((A, \sqsubseteq)\), and this is denoted by \(\alpha : C \rightleftarrows A : \gamma\) if all the following conditions hold:

- **\(\alpha\)-Monotonicity**: \(\forall y, y' \in C. y \subseteq y' \Rightarrow \alpha(y) \subseteq \alpha(y')\)
- **\(\gamma\)-Monotonicity**: \(\forall a, a' \in A. a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')\)
- **Galois**: \(\forall y \in C. y \subseteq \gamma(\alpha(y))\)
- **Connection**: \(\forall a \in A. \alpha(\gamma(a)) \sqsubseteq_A a\)

If the following condition, stronger than Connection, holds we say that \(\alpha\) and \(\gamma\) form a Galois Insertion and this is denoted also by \(\alpha : C \rightleftharpoons_A A : \gamma\)

- **Insertion**: \(\forall a \in A. \alpha(\gamma(a)) = a\)

Let \(\alpha_A : \wp(\text{Values}) \rightleftharpoons A : \gamma_A\) be a Galois insertion.

In the following we refer to \([30]\) (and to the references therein) for the formal proofs of correct constructions of complete lattices from other complete lattices using standard operators such as tupling, function spaces, lifting, etc.

When no confusion arises, we use the same meta-variables both for concrete and abstract objects. For instance, \(M, M'\) are meta-variables both for concrete memories and abstract ones. When needed, we add a superscript \# (or other specified notation) to abstract objects to distinguish them from the concrete ones.
5.1. Abstract memories

In Figure 7 it is shown the definition of the elements of the complete lattice

\[
\text{AbM}(A) = \{[1, 2, \ldots, N] \mapsto A \} \text{ ranged over by } M, M', \ldots
\]

and the definition of the functions of the Galois insertion

\[
\alpha_{MA}, \gamma_{MA} : \text{AbM}(A) \rightarrow \text{AbM}(A)
\]

\[
\text{AbM}(A) \text{ is a function space and its elements are called abstract memories on } A. \text{ The operators of the lattice are based on the operators of } (A, \sqsubseteq A) \text{ and they are constructed in the standard way for function spaces.}
\]

**Proposition 2.** \(\alpha_{MA} \text{ and } \gamma_{MA} \text{ form a Galois Insertion.}\)

The proof is in Appendix A.1.

5.2. Abstract stacks

The lattice of abstract stacks on \(A\),

\[
\text{AbSt}(A) = \{[1, 2, \ldots, N] \mapsto A \} \text{ ranged over by } St, St', \ldots
\]

and the Galois insertion

\[
\alpha_{StA}, \gamma_{StA} : \text{AbSt}(A) \rightarrow \text{AbSt}(A)
\]

are defined in Figure 8. Abstract stacks are composed by elements of \(A\) and can be compared, pointwise, only if they have the same height. To obtain a complete lattice we have added, by lifting, two new elements as top \((\top_{StA})\) and bottom \((\bot_{StA})\). They represent the lub and the glb, respectively, of every pair of abstract stacks having different height. We set their height as \(+\infty\) and \(-1\), respectively.

Note that a stack of the form \(\top_A \cdots \top_A \bot_A \cdots \bot_A\) is different from \(\top_{StA} \bot_{StA}\).

**Proposition 3.** \(\alpha_{StA} \text{ and } \gamma_{StA} \text{ form a Galois Insertion.}\)

The proof is in Appendix A.1.

5.3. Abstract states

Now we can easily define the abstract JVM states on \(A\) by pairing abstract memories and abstract stacks. The definition is in Figure 9.

\[
\text{FIGURE 7. The lattice and the Galois insertion for abstract memories.}
\]

**Proposition 4.** \(\alpha_{(MA, StA)} \text{ and } \gamma_{(MA, StA)} \text{ form a Galois insertion.}\)

**Proof.** The proof of all the properties can be simply derived, using the same arguments of the previous two proofs, from the definition and from the correspondent properties of \(\alpha_{MA}, \gamma_{MA}, \alpha_{StA} \text{ and } \gamma_{StA}.\)

5.4. Abstract domain

Finally, Figure 10 shows the definition of the abstract domain \(\text{AbDom}(A)\). The elements are all possible \(K\)-tuples\(^9\) of abstract states. The abstract state \(S_i\) of a tuple is associated to the bytecode program point \(i\) and represents the abstraction of a set of concrete JVM states at program point \(i\). Note that, if \(A\) is finite, then also \(\text{AbDom}(A)\) is finite.

Using this domain, every set of concrete JVM states \((i, (M, St))\) of the given bytecode program can be abstracted, by a suitable abstraction function, into a tuple of \(\text{AbDom}(A)\). This structure allows the abstract interpreter to perform correct abstract executions of \texttt{ret} instructions giving information about the state of all bytecode program points. This will become clearer in Section 5.6.

Recall that our abstract interpretation framework is able to handle different kinds of verifications. In [26] we use the framework to define the standard bytecode verifier. In the following section we define an abstract domain of types for the enhanced verifier.

5.5. Abstract types

Figure 11 shows a simple lattice \((D, \sqsubseteq D)\). The elements of \(D\) represent the possible types of a register or of a stack element of a JVM state. Element \(\bot_D\) represents an empty set of values. It will be the value of every register in an abstract state \(S_i\), if no predecessor of instruction \(i\) has ever been executed in the abstract computation leading to \(S_i\), for example the value of the registers in the initial state of each instruction different from the first one. The type \texttt{unt} abstracts instead the undefined value, that is the contents of a not assigned

\(^9\)Recall that \(K\) is the number of instructions of the given bytecode program.
An Abstract Interpretation Approach for Enhancing the Java Bytecode Verifier

\[
\begin{align*}
\text{AbSt}(A) &= \{(\bot_{\text{St}}, \top_{\text{St}}) \cup \{s_1, s_2, \ldots, s_p \mid 1 \leq p \leq \text{max stack height}, \forall i, 1 \leq i \leq p, s_i \in A\} \\
\subseteq_{\text{St}} &= (\bot_{\text{St}}, \top_{\text{St}}) \subseteq_{\text{St}} s'_i \ldots s'_p \text{ if } \forall j \in \{1, 2, \ldots, p\}. s_i \subseteq_{A} s'_i \\
\sqcup_{\text{St}} &= (\bot_{\text{St}}, \top_{\text{St}}) \sqcup_{\text{St}} s'_i \ldots s'_p = (s_1 \cup_{A} s'_1) \ldots (s_p \cup_{A} s'_p) \\
\sqcap_{\text{St}} &= (\bot_{\text{St}}, \top_{\text{St}}) \sqcap_{\text{St}} s'_i \ldots s'_p = \top_{\text{St}} \text{ if } p \neq h \\
\sqcap_{\text{AbDom}} &= (\bot_{\text{AbDom}}, \top_{\text{AbDom}}) \sqcap_{\text{AbDom}} (s_1, s_2, \ldots, s_p) = (s_1 \sqcap_{A} s'_1) \ldots (s_p \sqcap_{A} s'_p) \\
\top_{\text{St}} &= \forall s \in \text{AbSt}(A). \exists s \subseteq_{\text{St}} s \top_{\text{St}} \\
\alpha_{\text{St}} &= \alpha_{\text{St}}(\emptyset) = \bot_{\text{St}} \\
\alpha_{\text{AbDom}} &= \alpha_{\text{AbDom}}(\{M, St \mid M \subseteq M_{A} \} \cap \{St \mid St \subseteq St\}) \\
\gamma_{\text{St}}(\top_{\text{St}}) &= \text{Stack(Values)} \\
\gamma_{\text{St}}(\bot_{\text{St}}) &= \emptyset \\
\gamma_{\text{St}}(s_1, s_2, \ldots, s_p) &= \{s_1, s_2, \ldots, s_p \mid s_i \subseteq_{A} s'_i, s_i \in \text{AbSt}(A)\}
\end{align*}
\]

**FIGURE 8.** The lattice and the Galois insertion for abstract stacks.

\[
\begin{align*}
\text{AbState}(A) &= \text{AbM}(A) \times \text{AbSt}(A) \text{ ranged over by } (M, St) \text{ or } S_1, S_2, \ldots \\
\subseteq_{\text{M, St}} &= (M, St) \subseteq_{\text{M, St}} (M', St') \text{ if } M \subseteq_{M_{A}} M' \text{ and } St \subseteq_{St} St' \\
\sqcup_{\text{M, St}} &= (M, St) \sqcup_{\text{M, St}} (M', St') = (M \sqcup_{M_{A}} M', St \sqcup_{St} St') \\
\sqcap_{\text{M, St}} &= (M, St) \sqcap_{\text{M, St}} (M', St') = (M \sqcap_{M_{A}} M', St \sqcap_{St} St') \\
\top_{\text{M, St}} &= (\top_{M_{A}}, \bot_{St}) \\
\alpha_{\text{M, St}} &= \alpha_{\text{M, St}}(M, St) \subseteq_{\text{M, St}} (M', St') = (\alpha_{M_{A}}(M, St) \subseteq_{M_{A}} M'_{A}), \alpha_{\text{St}}(St \subseteq_{\text{St}} St') \\
\gamma_{\text{M, St}} &= \gamma_{\text{M, St}}((M, St)) \subseteq_{\text{M, St}} (\{z \mid z \in \gamma_{\text{M}_{A}}(M, St), y \in \gamma_{\text{St}}(St)\})
\end{align*}
\]

**FIGURE 9.** The lattice and the Galois insertion for abstract JVM states.

\[
\begin{align*}
\text{AbDom}(A) &= \{(S_0, S_1, \ldots, S_{K-1}) \mid S_i \in \text{AbState}(A)\} \text{ ranged over by } S, S', \ldots \\
\subseteq_{\text{AbDom}} &= S \subseteq_{\text{AbDom}} S' \text{ if } \forall i \in \{0, 1, \ldots, K-1\}. S_i \subseteq_{\text{M, St}} S'_i \\
\sqcup_{\text{AbDom}} &= S \sqcup_{\text{AbDom}} S' = (S_0 \sqcup_{\text{M, St}} S'_0, S_1 \sqcup_{\text{M, St}} S'_1, \ldots, S_{K-1} \sqcup_{\text{M, St}} S'_{K-1}) \\
\sqcap_{\text{AbDom}} &= S \sqcap_{\text{AbDom}} S' = (S_0 \sqcap_{\text{M, St}} S'_0, S_1 \sqcap_{\text{M, St}} S'_1, \ldots, S_{K-1} \sqcap_{\text{M, St}} S'_{K-1}) \\
\top_{\text{AbDom}} &= (\top_{\text{M, St}}, \ldots, \top_{\text{M, St}}) \\
\bot_{\text{AbDom}} &= (\bot_{\text{M, St}}, \ldots, \bot_{\text{M, St}})
\end{align*}
\]

**FIGURE 10.** The abstract domain.
register. The unique concrete value represented by this type is \((\omega, \Omega)\). \(\top_D\) represents a contradictory type (resulting from the merge of incompatible types). For each type \(\tau\) (different from \(\top_D\) and \(\bot_D\)), the type \(\text{unt}_\tau\) means that the register, at a merge point, has been assigned a value of type \(\tau\) on some program paths and has not been touched on the other paths; in terms of values this means that the type \(\text{unt}_\tau\) represents all the values represented by \(\tau\) and also the value \((\omega, \Omega)\). The type \(\text{int}\) represents integers and a type \(\text{ret}(L)\) represents the set of values \(\{(\ell, \text{ret}(L)) \mid \ell, L \in \{0, 1, \ldots, K - 1\}, \ell > 0, P(\ell - 1) = \text{jsr } L\}\), i.e., the addresses of all the return points from a subroutine starting at \(L\).

Figure 12 shows the definition of the Galois insertion 
\[
\alpha_D: \varphi(\text{Values}) \subseteq D: \gamma_D.
\]
As usual in defining abstractions, the abstraction of a set of values \(y\) is the best (the more precise in the abstract domain) abstract value that can represent \(y\). Here this is simply expressed by defining \(\alpha_D(y)\) as the gbl of a set \(\preceq\alpha_D\) containing abstractions of \(y\) at different levels of precision. The set \(\preceq\alpha_D(y)\), where \(y \subseteq \text{Values}\), is defined as follows:

- \(\top_D \in \preceq\alpha_D(y)\)
- \(\text{unt_int} \in \preceq\alpha_D(y)\) iff \(y \subseteq \{(\omega, \Omega)\} \cup \mathbb{Z} \times \{\text{int}\}\)
- \(\text{int} \in \preceq\alpha_D(y)\) iff \(y \subseteq \mathbb{Z} \times \{\text{int}\}\)
- \(\text{unt_ret}(L) \in \preceq\alpha_D(y)\) iff \(y \subseteq \{(\ell, \text{ret}(L)) \mid \ell \in \{0, 1, \ldots, K - 1\}, \ell > 0, P(\ell - 1) = \text{jsr } L\} \cup \{(\omega, \Omega)\}\)
- \(\text{ret}(L) \in \preceq\alpha_D(y)\) iff \(y \subseteq \{(\ell, \text{ret}(L)) \mid \ell \in \{0, 1, \ldots, K - 1\}, \ell < 0, P(\ell - 1) = \text{jsr } L\}\)
- \(\bot_D \in \preceq\alpha_D(y)\) iff \(y = \emptyset\)

Using the construction defined in Section 5, we can obtain, starting from \((D, \subseteq D)\), the complete lattice

\[
\left(\text{AbDom}(D), \subseteq\text{AbDom}(D), \sqcup\text{AbDom}(D), \sqcap\text{AbDom}(D), \top\text{AbDom}(D), \bot\text{AbDom}(D)\right).
\]

The construction of Section 5 provides, starting from the Galois insertion \(\alpha_D: \varphi(\text{Values}) \subseteq D: \gamma_D\), a Galois insertion

\[
\alpha(M_A, S_{A}) \varphi(\text{Mem}(\text{Values}) \times \text{Stack}(\text{Values})) \subseteq \text{AbState}(D): \gamma(M_A, S_{A})
\]

between the sets of concrete JVM states and the abstract JVM states based on \(D\).

In Figure 13 we define a Galois connection:

\[
\alpha_{\text{AbDom}(D)}: \varphi(\text{States}) \subseteq \text{AbDom}(D): \gamma_{\text{AbDom}(D)}
\]

Note that we introduce, for the definition of the abstraction function, a predicate \(\text{Cons}_D\) on sets of concrete bytecode program states. This predicate says, when abstracting a set \(y\), whether \(y\) is consistent w.r.t. the structure of the bytecode program. If not, the abstraction of the set is the top element of \(\text{AbDom}(D)\), otherwise the abstraction is done - abstract state by abstract state - by \(\alpha(M_A, S_{A})\), to form a \(K\)-tuple.

The consistency conditions expressed by \(\text{Cons}_D\) check some properties of the values of the registers at \(\text{ret}\) instructions w.r.t. their values at calling points, taking into account which registers are modified during the execution of the subroutine.

In order to define precisely these conditions we have to specify some operations on the structure of the given bytecode program.

Every \(\text{ret} x\) instruction has several successors, namely all program points that follow a \(\text{jsr } L\) instruction jumping to the subroutine starting at \(L\), which is returned by the \(\text{ret}\). We denote this set by \(\text{Static_Return_Points}(L)\). Moreover we use \(\text{MR}(L)\) to denote the set of registers that are modified within the code of the subroutine that starts at \(L\) and \(\text{SubStart}(i)\), where \(P(i) = \text{ret } x\), to denote the program point \(L\) at which the subroutine which is returned starts.

The effective calculation of \(\text{MR}\) may not be simple, due to the lack of structure of the bytecode. Detailed discussions on this topic can be found in [27, 10, 8]. We remark that our approach is independent of the way this calculation is done.

In Appendix A.2 we describe a labelling procedure, analogous to the one defined by Wildmoser in [27], that shows how the calculation can be carried out by a preliminary abstract interpretation on the code. We state minimal reasonable assumptions on the code that we need to do the calculation. A program that does not satisfy these minimal assumptions cannot be verified and thus we assume that the verifier rejects it without performing any verification. In particular, we assume that each subroutine has a single entry point reachable by a \(\text{jsr}\) instruction, and that each entry point contains an \(\text{astore}\) instruction. We assume also that \(\text{astore}\) instructions at different entry points act on different registers and, finally, we do not allow recursive subroutine calls.

The condition expressed by the predicate \(\text{Cons}_D\) is needed to maintain the Galois condition of the
The third condition of the predicate $\text{Cons}_D(y)$ requires that, at a $\text{ret}$ program point, the registers modified by the subroutine hold the value $(\omega, \Omega)$ only if it was present at their actual calling point. Recall that the value $(\omega, \Omega)$ represents the contents of uninitialised registers and it is clearly impossible in any concrete execution that such value is present at some register in some program point if it was not present in the same register in a preceding point in the computation. We check this only between the calling $\text{jsr}$ points and the returning $\text{ret}$ point. The former are obviously preceding points in the computation of the latter.

A set of states violating the requirements expressed above can not be generated by any concrete computation and thus we abstract it to $\uparrow_{\text{AbDom}(D)}$.

It is important to say, at this point, that the second and the third conditions of the predicate $\text{Cons}_D$ have been introduced to assure the local correctness of the abstract next operator defined in the following section. There, the reasons of this need can be viewed more clearly and formally especially in the proof of the

\begin{align*}
\alpha_D(y) &= \left\{ \begin{array}{ll}
\text{Values} & \text{if } \tau = \top_D \\
\{\omega, \Omega\} \cup \mathbb{Z} \times \{\text{int}\} & \text{if } \tau = \text{unt_int} \\
\{\omega, \Omega\} & \text{if } \tau = \text{unt} \\
\mathbb{Z} \times \{\text{int}\} & \text{if } \tau = \text{int} \\
\end{array} \right.
\end{align*}

\begin{align*}
\gamma_D(\tau) &= \left\{ \begin{array}{ll}
\{\omega, \Omega\} \cup \{\ell, \text{ret}(L)\} & \text{if } \tau = \text{unt_ret}(L) \\
\{\ell, \text{ret}(L)\} & \text{if } \tau = \text{ret}(L) \\
\emptyset & \text{if } \tau = \bot_D \\
\end{array} \right.
\end{align*}

\begin{align*}
\alpha_{\text{AbDom}(D)}(y) &= \left\{ \begin{array}{ll}
\uparrow_{\text{AbDom}(D)}(D) & \text{if } -\text{Cons}_D(y) \\
\alpha_{(M_D, St_D)}(\{(M, St) \mid (0, (M, St)) \in y\}), & \\
\alpha_{(M_D, St_D)}(\{(M, St) \mid (1, (M, St)) \in y\}), & \\
\quad \ldots & \\
\alpha_{(M_D, St_D)}(\{(M, St) \mid (K - 1, (M, St)) \in y\}) & \text{if } \text{Cons}_D(y)
\end{array} \right.
\end{align*}

\begin{align*}
\gamma_{\text{AbDom}(D)}(S) &= \left\{ \begin{array}{ll}
\varphi(\text{States}) & \text{if } S = \uparrow_{\text{AbDom}(D)} \\
\bigcup_{i=0}^{K-1} \{ (i, \sigma) \mid P(i) \neq \text{ret}, x, \sigma \in \gamma_{(M_D, St_D)}(S_i) \} \cup \\
\bigcup_{i=0}^{K-1} \{ s \in \varphi(\text{States}) \mid P(i) = \text{ret}, \text{SubStart}(i) = L, \}
\quad s \in \text{Ret}_\text{Concr}_D(i, L, S, x) & \text{if } S \neq \uparrow_{\text{AbDom}(D)}
\end{array} \right.
\end{align*}
Proposition 7 where the correctness of the abstract rule for the ret instruction needs the consistency of the concrete states expressed by ConsD.

The concretization function γAbDom(D) defined in Figure 13 is designed to respect the consistency condition of the abstraction function. It must not concretize consistent abstract states into inconsistent sets of concrete states. This has been done by using the operator RetConcrD, which is defined in Figure 15.

It is called with i such that P(i) = ret x and L = SubStart(i). The result of the operator depends on the values of the abstract JVM states St with ℓ ∈ Static_Return_Points(L) that are different from \( \downarrow(MD, StD) \) and \( \uparrow(MD, StD) \). This means that every generated concrete state associated to a ret x instruction contains only correct return points. Moreover note that, by definition, \( \forall S_i = (M_i, S_i) \in AbState(D) \). P(i) = ret x ∧ SubStart(i) ⇒ RetConcrD(i, L, S, x) ⊆ \( \gamma_{MD, StD}(S_i) \). This means that the concretization of the abstract states at ret instructions is the “natural” set \( \gamma_{MD, StD}(S_i) \) minus the inconsistent concrete states that may be generated by the loss of information, at the abstract level, on the exact return point for each concrete state.

The following Lemma states two properties, one of the predicate ConsD and one of the operator RetConcrD, that will be useful for proving the monotonicity of the abstraction and concretization functions of the Galois connection defined for the abstract domain AbDom(D).

**Lemma 1.** Let \( y, y' \in \phi(States) \) such that \( y \subseteq y' \) and let \( S = (S_0, S_1, \ldots, S_K) \), \( S' = (S'_0, S'_1, \ldots, S'_K) \) \( \in AbDom(D) \) such that \( S \subseteq_{AbDom(D)} S' \).

The following properties hold:
(i) \( \neg\text{ConsD}(y) \Rightarrow \neg\text{ConsD}(y') \)
(ii) \( \forall \ell, L, i \in \{0, \ldots, K-1\}, \forall x \in \{1, \ldots, N\}, \{s \in \phi(States) \mid P(i) = \text{ret x}, \text{SubStart}(i) = L, s \in \text{RetConcrD}(i, L, S, x) \} \subseteq \{s \in \phi(States) \mid P(i) = \text{ret x}, \text{SubStart}(i) = L \} \).
unloadable and this situation does not necessarily imply an error.

Let us describe the rules. Instruction \texttt{iconst}# pushes a constant value onto the stack. The abstract stack of the successor instruction \(i + 1\) is constructed by pushing the type \texttt{int} onto the abstract stack of instruction \(i\); it is required that the push does not overflow the stack. In the subsequent instruction (with index \(i + 1\)) abstract memory and stack are assigned to the least upper bound of the old values and the ones coming from the abstract execution of \(i\). If the state \(S_{i+1}\), at the program point \(i + 1\), has not been considered yet in the computation (i.e. \(S_i = \bot_{(M_D, St_D)}\)), then \(S_{i+1} \circlearrowleft (M_D, St_D)\) \((M, \text{int}\text{.}S)\) simply results in \((M, \text{int}\text{.}S)\). \texttt{iload}# requires that the loading register has type \texttt{int}, and pushes \texttt{int} onto the stack.

Instructions \texttt{istore}# and \texttt{astore}# store an integer or an address, respectively, from the stack to a register. \(M[^{i}/x]\) denotes the usual modification of the function \(M\) at point \(x\) with the new value \(v\).

Instruction \texttt{ifeq}# requires an integer on the stack. The abstract interpreter executes both branches of conditionals and updates both successor program points \((L\text{.}i+1)\), respectively, from the stack to a register. \(M[^{i}/x]\) denotes the usual modification of the function \(M\) at point \(x\) with the new value \(v\).

Instruction \texttt{ifeq}# requires an integer on the stack. The abstract interpreter executes both branches of conditionals and updates both successor program points \((L\text{.}i+1)\), respectively, from the stack to a register. \(M[^{i}/x]\) denotes the usual modification of the function \(M\) at point \(x\) with the new value \(v\).

In order to compute the state of the return points from a subroutine the \texttt{ret}# rule uses the \(\triangleright_L\) operator, which acts as follows

\[
\begin{align*}
(M, St) \triangleright_L (M', St') &= (M \triangleright_L M', St') \\
\triangleq_{M_D} M \triangleright_L M' &= \triangleq_{M_D} \\
(M \triangleright_L M')(x) &= \begin{cases} M(x) \triangleright M'(x) & \text{if } x \in \text{MR}(L) \\ M(x) & \text{if } x \notin \text{MR}(L) \end{cases}
\end{align*}
\]

The difference with respect to the standard verifier is that the new value of the registers modified by the subroutine depends also on the value of the register at calling points. The value of a modified register \(x\) at a return point \(\ell\) from a subroutine is \(M_{\text{ret}}(x) \triangleright M_\ell(x)\) where \(M_{\text{ret}}\) is the abstract memory at the \texttt{ret}# instruction. Figure 16 shows the definition of the \(\triangleright\) operator between elements of \(D\). Each entry of the table contains the type resulting from the type on the row \(\triangleright\) the type on the column. Note that the operator is not commutative. The type \(\tau_1 \triangleright \tau_2\) is the type that a modified register, having type \(\tau_1\) at the calling \texttt{jsr}# instruction and type \(\tau_2\) at the \texttt{ret}# program point, should have - after the \texttt{ret}# to achieve a correct typing. We remark that \(\triangleright\) \texttt{unt}\_\texttt{int} = \texttt{int}\) and this definition allows us to solve the subroutine problem, as explained before. If we have two types, \(\texttt{ret}(L_1)\) and \(\texttt{ret}(L_2)\) with \(L_1 \neq L_2\), they are considered incompatible, hence, for example, it would be \(\texttt{ret}(L_1) \triangleright \texttt{unt}\_\texttt{ret}(L_2) = \top\).

Note that, as we previously observed, the abstract execution of the \texttt{ret}# instruction requires information about the abstract states of program points different from the one of the \texttt{ret}# instruction that is executed. This need motivates the definition of the abstract domain as a set of \(K\)-tuples of abstract JVM states.

Finally note that, at all return points, \(\top\) is assigned to the register \(x\) of the \texttt{ret}# instruction. This prevents the use of a return address more than once, while it is intended to be used only for one returning.

In addition to the rules of Figure 17 there are several rules for managing error situations. These rules simply update the successors abstract states of the original rule with \(\top_{(M_D, St_D)}\). For instance, one of the error rules for the rule \texttt{ifeq}# is the following:

\[
\begin{align*}
P(i) &= \texttt{ifeq} L, S_i = (M, St) \neq \bot_{(M_D, St_D)}, \\
M_\triangleright_{(M_D, St_D)} &= \top_{St_D} \\
\forall i+1 \in \{0, 1, \ldots, K - 1\}, \quad L \in \{0, 1, \ldots, K - 1\}
\end{align*}
\]

Note that an error propagates from the program point in which it occurs to all the program points that are reachable from that. This type of error handling guarantees the correctness of the abstract next operator w.r.t. the concrete next operator. For practical purposes, the algorithm of verification can be defined such that it rejects the code if it executes at least one error rule.

Now we can define precisely the abstract step operator:
\textbf{Definition 5.2. (Abstract next operator)} Let \( S \in \text{AbDom}(D) \) be an abstract state. An abstract computation step is modeled by the following operator \( \text{next}^{\text{Ab}(D)} : \text{AbDom}(D) \rightarrow \text{AbDom}(D) \)

\[
\text{next}^{\text{Ab}(D)}(S) = \begin{cases} 
\top_{\text{AbDom}(D)} & \text{if } S = \top_{\text{AbDom}(D)} \\
\sqcup_{\text{AbDom}(D)} \{ S' \in \text{AbDom}(D) | S \neq \top_{\text{AbDom}(D)} \} & \text{if } S \neq \top_{\text{AbDom}(D)} 
\end{cases}
\]

Note that the result of an abstract step is the hub of all possible states derived from all possible transitions \( \rightarrow^{\text{AbDom}} \) from a state. Thus, every state is recalculated using the information added on the previous step, until no new information can be added.

\textbf{Proposition 6. (Monotonicity)} \( \text{next}^{\text{Ab}(D)} \) is a monotone operator in the lattice \((\text{AbDom}(D), \sqsubseteq_{\text{AbDom}(D)})\): \( \forall S, S' \in \text{AbDom}(D) \)

\[ S \sqsubseteq_{\text{AbDom}(D)} S' \implies \text{next}^{\text{Ab}(D)}(S) \sqsubseteq_{\text{AbDom}(D)} \text{next}^{\text{Ab}(D)}(S') \]

The proof is in Appendix A.1.

\textbf{Proposition 7. (Local Correctness)} \( \text{next}^{\text{Ab}(D)} \) is a correct abstraction of the concrete \( \text{next}^{\text{C}} \), i.e.,
\[
\forall y \in \varphi(\text{States}), \text{next}^{\text{C}}(y) \sqsubseteq \gamma_{\text{AbDom}(D)}(\text{next}^{\text{Ab}(D)}(\alpha_{\text{AbDom}(D)}(y)))
\]

The proof is in Appendix A.1.

\textbf{Definition 5.3. (Abstract Semantics)} Given a bytecode program with initial state \((0, (M_0, ()))\), its Abstract Semantics calculated by the enhanced abstract interpreter, based on the domain \( D \), is called \( \text{AbSem}(D) \) and it is the least upper bound, in \( \text{AbDom}(D) \), of the increasing chain defined, for all \( n \in \mathbb{N} \), as follows:

\[
\text{AbSem}(D)^0 = \alpha_{\text{AbDom}(D)}(\{(0, (M_0, ()))\})
\]
\[
\text{AbSem}(D)^{n+1} = \text{next}^{\text{Ab}(D)}(\text{AbSem}(D)^n)
\]

The abstraction of all possible initial concrete states \( S \) with \( S_i = \bot_{(M_D, StD)} \) for all

\[ i = 1, 2, \ldots, K - 1 \text{ and } S_0 = (M_0, ()) \text{ such that } M_0(x) = \text{unt} \text{ for all } x \in \{ m+1, m+2, \ldots, N \}. \]

The first \( m \) positions of the abstract memory \( M_0 \) contain the types of the values indicated in the signature of the method. Since \( \text{AbDom}(D) \) is a finite domain and the step is monotonic the computation of the abstract semantics always terminates.

We conclude this section by formally expressing the fact that \( \text{AbSem}(D) \) is a correct approximation of the accumulating semantics \( C \) of the bytecode program.

\textbf{Theorem 5.1. (Global Correctness)} Given a bytecode program, let \( C \) be its accumulating semantics, defined in Definition 4.3, and let \( \text{AbSem}(D) \) be its abstract semantics, based on the domain \( D \). Then,

\[ \alpha_{\text{AbDom}(D)}(C) \sqsubseteq_{\text{AbDom}(D)} \text{AbSem}(D) \]

or, equivalently,

\[ C \sqsubseteq \gamma_{\text{AbDom}(D)}(\text{AbSem}(D)) \]

Proof. By Propositions 6 and 7 and by the general theorems of abstract interpretation [22, 24].

6. NOTES ON IMPLEMENTATION

In this section we want to underline some aspects that may arise in an effective implementation of the abstract interpretations we have defined.

First of all, as we show in [31], the enhanced abstract execution can be implemented in the standard way, i.e. using a worklist algorithm performing, at each step, an abstract next \(( \rightarrow^{\text{AbDom}} \) ) transition on a program point \( i \) belonging to the worklist. The program points that have changed after a step are inserted in the worklist. At the beginning the worklist contains only the first instruction of the bytecode program with the initial abstract memory and stack. The algorithm stops when the worklist becomes empty. This is the algorithm implemented in many existing verifiers.

For efficiency reasons, it is required that the body of the subroutine is not checked separately for the different calling points. This can be implemented easily by a suitable selection rule from the worklist.

We want to remark that this algorithm is defined independently from the abstract states and from the corresponding abstract next operator that are used. Thus, any implementation of the standard verification

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\text{Oper}
 & \text{unt}
 & \text{int}
 & \text{ret}(l)
 & \text{unt_int}
 & \text{unt_ret}(l)
 & \top_D
\hline
\top_D
 & \text{unt}
 & \text{int}
 & \text{ret}(l)
 & \text{unt_int}
 & \text{unt_ret}(l)
 & \top_D
\hline
\top_D
 & \text{int}
 & \text{int}
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
\hline
\top_D
 & \text{ret}(l)
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
 & \top_D
\hline
\top_D
 & \text{unt_int}
 & \text{int}
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
\hline
\top_D
 & \text{int}
 & \text{int}
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
\hline
\top_D
 & \text{ret}(l)
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
 & \top_D
\hline
\top_D
 & \text{unt_int}
 & \text{int}
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
\hline
\top_D
 & \text{ret}(l)
 & \text{ret}(l)
 & \text{int}
 & \top_D
 & \top_D
 & \top_D
\hline
\top_D
 & \text{unt_ret}(l)
 & \text{unt_ret}(l)
 & \text{int}
 & \top_D
 & \top_D
 & \top_D
\hline
\end{tabular}
\caption{Full definition of the \textgreater operator.}
\end{table}
can be easily updated to our enhanced verification simply changing the abstract domain and the behaviour at \texttt{ret} instructions. It is important to note that the efficiency of the algorithm is not affected at all by the updating. However, a preliminary step for detecting subroutine boundaries has to be added.

Our approach can be extended to the full JVML by defining the abstract rules for all the instructions not considered in this paper and by expanding the domain \( D \) properly. The definition of the rules can be done quite easily following the descriptions of [1]. Regarding the domain, we have to enrich \( D \) with the other JVM types - i.e. \texttt{float}, object references, arrays, \texttt{long} and \texttt{double} - ensuring that it remains a lattice in the process.

Introducing object references could be difficult because the source-level types together with the subtyping relation do not form a lattice. This derives from the fact that interfaces are types and classes can implement several interfaces. However, several solutions have been presented in the literature to overcome this problem. They are referred in [10].

7. CONCLUSIONS AND RELATED WORKS

We have developed an abstract interpretation framework to analyse Java bytecode programs. Within this framework, we have defined an enhanced verifier to address some limitations of the standard verifier when
dealing with subroutines. We have improved the verification by providing a richer abstract domain (more precise in the standard sense of abstract interpretation) than the one used for standard verification and a different behaviour at \texttt{ret} instructions. This yields an enhanced verification that is more accurate and that accepts a larger set of type-safe bytecode programs than the standard one.

We remark that the definition of the concrete domain and the construction of the abstract domains we have provided represent a clear and formal basis for other static analyses of bytecode programs.

It is important to note that our approach verifies a set of correctly compiled methods which strictly includes the set of the ones accepted by the standard verifier. This does not mean that all the correctly compiled methods are accepted by using the proposed verification approach. To state this result, which is out of the scope of the current work, we should prove that the static analysis performed by the Java compiler is included in our abstract interpretation. To do this we would need a formal definition of the analysis performed by the compiler. Then the proof could be done by embodying such an analysis within our abstract interpretation framework. A result of this type can be found in [19] where a notion of compilation is formalised and it is shown that the verifier defined therein accepts all compiled programs.

In this paper we have concentrated on the problem of subroutines and thus we have considered a restricted set of JVM instructions and types, but it should be possible to treat the whole JVML with all types by simply adding the concrete and abstract rules for the instructions and specifying the basic type domain as a complete lattice including, for each basic or class type \( \tau \) the type \texttt{unt}_{\tau}.

The official definition of the bytecode verifier consists of an informal specification of its features and of an implementation [1]. Such definition is not satisfactory for formal reasoning and checking purposes. Thus, many formalisations of (parts of) the JVML and of the verifier have been introduced using different techniques. During this task, several bugs or inconsistencies have been pointed out, such as the problem considered in this paper.

In [2] the same fragment of JVML considered in this paper is used to define a type system modelling the standard verifier and to formally prove its soundness. In [15] another type system for the same fragment is defined. It improves the previous one allowing to safely type a larger class of bytecode programs. A sound type system modelling the standard verifier is also defined in [3]. It treats a larger set of JVML instructions than the previous ones. A mathematical specification of the standard verifier as a data-flow analysis architecture can be found in [4]. In [16] it is proposed an extension of the type systems in [15, 3] to treat concurrency and mutual exclusion issues.

In [5] also our fragment of JVML is considered and the standard verifier is modeled by a finite chaotic fixpoint computation. This approach is close to ours, but we avoid the problems of non-monotonicity raised in modelling subroutines by using structural properties of the bytecode programs as for the operator \( \text{HR}(L) \).

An interesting approach to the formalisation of the JVM and of the bytecode verifier is the use of theorem provers. In [6, 11] a large part of the JVM and of the verifier has been modeled within the theorem prover Isabelle/HOL. Then the soundness of the verifier has been formally proved by the tool. In [7] the theorem prover Coq is used to implement a type system defining the verifier and to get a mechanical proof of soundness as well as a direct implementation of the verifier in CAML.

Let us now compare our approach with others solving the subroutine problem. The work [10] solves the problem for a bytecode verifier for Java Card\footnote{It also applies to Java.} by defining a polyvariant approach where subroutines are examined in different contexts for different callers. The approach is less efficient than the standard bytecode verifier but it is feasible since the verifier acts off-card. Also the verifier proposed in [32], based on model checking, is meant for an off-line verification. The work [8] proposes a solution based on the use of sets of types instead of types. The idea is to consider for each register and for the stack the types for each different call, where these types are annotated with the calling address, in order to be taken distinct. In this way a polyvariant verification of subroutines is possible and subroutines are not managed in a special way with respect to the rest of the code. The main advantage of this approach is that it is not necessary to compute the subroutine boundaries and the set of the registers which are modified inside the subroutine. However, the verification algorithm is different from the standard one, since it handles sets of annotated types and performs the analysis in all possible cases. Hence, it is in general less efficient than the standard verifier, even if in many cases the performance of the two algorithms is comparable.

We remark that the above cited works do not fit in our framework, since they follow an algorithm which is different from that of the standard verifier. In fact we rely for subroutine verification on standard bytecode verification and the only extension we perform is enriching the domain of types. Moreover, we do not propose a more powerful verification technique, allowing to certify recursive or polymorphic subroutines. We fix a problem of the standard verifier which has not been solved by other extensions without changing the standard verification algorithm.
A.1. PROOFS OF THE PROPOSITIONS

In this section we give the proofs of the technical propositions regarding the verification of the properties needed to build the abstract interpretation framework.

Proof of Proposition 2.

Proof. (α-Monotonicity).
Let \( y, y' \in p(Mem(\text{Values})) \) and let \( x \in \{1, 2, \ldots, N\} \).
\( y \subseteq y' \Rightarrow \{M(x) \mid M \in y\} \subseteq \{M(x) \mid M \in y'\} \).
Thus, by monotonicity of \( \alpha_A, \alpha_A(\{M(x) \mid M \in y\}) \subseteq \alpha_A(\{M(x) \mid M \in y'\}) \).
By definition of \( \alpha_M \), this entails \( \alpha_M(y)(x) \subseteq \alpha_M(y')(x) \).

(γ-Monotonicity). Let \( M, M' \in Ab^\gamma(A) \).
\( M \subseteq_M M' \equiv \forall x \in \{1, 2, \ldots, N\}, \{M(x) \mid M \in y\} \subseteq \{M(x) \mid M \in y'\} \).
By monotonicity of \( \gamma_A \), we get that \( \forall x \in \{1, 2, \ldots, N\}, \gamma_A(M(x)) \subseteq \gamma_A(M'(x)) \).
By definition of \( \gamma_M \), this implies \( \gamma_M(M') \subseteq \gamma_M(M') \).

(Galois). Let \( y \in p(Mem(\text{Values})) \) and let \( M \in y \).
By monotonicity of \( \gamma_M \), we have that \( \{M \mid \gamma_M(M'(x)) \subseteq \gamma_M(M(x)) \} \subseteq \gamma_M(M(y)) \).
By monotonicity of \( \gamma_M \), this implies \( \gamma_M(\alpha_M(M'(x))) \subseteq \gamma_M(\alpha_M(M(x))) \).
By definition of \( \alpha_M \), we have that \( \forall x \in \{1, 2, \ldots, N\}, \alpha_M(M'(x)) = \alpha_M(M(x)) \).
Since \( (\alpha_A, \gamma_A) \) forms a Galois Insertion, we know that \( \forall x \in \{1, 2, \ldots, N\}, \{M(x) \mid M \in y\} \subseteq \gamma_A(\alpha_M(M(x))) \).
Thus, by the definition of \( \gamma_M \), we can conclude that \( M \in \gamma_M(\alpha_M(M'(x))) \) and, by the previous inclusion, that \( M \in \gamma_M(\alpha_M(M(y))) \).

Proof of Proposition 3

Proof. (α-Monotonicity).
Let \( y, y' \in p(Stack(\text{Values})) \) such that \( y \subseteq y' \). If \( y \) is empty then, by definition, \( \alpha_M(\emptyset) = \perp_M \), which is trivially less than \( \alpha_M(y') \). If \( y' \) is empty, then also \( y \) must be empty by hypothesis, thus the previous case applies. If \( \alpha_M(\emptyset) = \perp_M \), then also \( \alpha_M(y') = \perp_M \) and, trivially, \( \alpha_M(y) \subseteq \alpha_M(y') \).
If \( \alpha_M(\emptyset) = \perp_M \) and \( \alpha_M(y') = \perp_M \) then the thesis is trivially true.

Suppose \( \alpha_M(y) \neq \top_M \) and \( \alpha_M(y') \neq \top_M \). Then, by definition of \( \alpha_M \), we can say that \( \exists p \in N : \forall \text{St}, \text{St}' \in \{y \cup y', \text{St} = \text{St}' = p\} \). Thus, \( y \subseteq y' \Rightarrow \forall j \in \{1, 2, \ldots, p\}, \{s_j \mid s_1, s_2, \ldots, s_p \in y\} \subseteq \{s_j \mid s_1, s_2, \ldots, s_p \in y\} \).
By monotonicity of \( \alpha_A \), that \( \forall j \in \{1, 2, \ldots, p\}, \alpha_A(\{s_j \mid s_1, s_2, \ldots, s_p \in y\}) \).
By definition of \( \alpha_M \) and of \( \alpha_M \), this entails \( \alpha_M(y)(y) \subseteq \alpha_M(y')(y) \).

(γ-Monotonicity). Let \( \text{St}, \text{St}' \in Ab^\gamma(A) \) such that \( \text{St} \subseteq_M \text{St}' \).
Again, \( \text{St} = \perp_M \) then, by definition, \( \gamma_M(\perp_M) = \emptyset \), which is trivially included in \( \gamma_M(\perp_M) \). If \( \text{St}' = \perp_M \), then also \( \text{St}' = \perp_M \) by hypothesis, thus the previous case applies, if \( \text{St} = \top_M \) then also \( \text{St}' = \top_M \) and obviously the thesis is true. If \( \text{St} \neq \top_M \) and \( \text{St}' = \top_M \) then, whatever \( \text{St} \) is, its concretization is surely a subset of \( \gamma_M(\top_M) \) because this contains all possible concrete stacks.
Suppose \( \text{St} \neq \top_M \) and \( \text{St}' \neq \top_M \). Then it must be that \( [\text{St}] = [\text{St}'] = p \) (for some \( p \in N \)) because, if they were of different length, they would be unrelated by \( \subseteq_M \). In these hypotheses, \( \text{St} \subseteq_M \text{St}' \) implies, by definition of \( \subseteq_M \), that \( \text{St} = s_1, s_2, \ldots, s_p \wedge \text{St}' = s_1', s_2', \ldots, s_p' \wedge \forall j \in \{1, 2, \ldots, p\}, s_j \subseteq_M s_j' \).
By monotonicity of \( \gamma_M \), this entails that \( \forall j \in \{1, 2, \ldots, p\}, \alpha_M(s_j) \subseteq \alpha_M(s_j') \).
Thus, by Cartesian product definition, we have that \( \gamma_M(s_1) \times \cdots \times \gamma_M(s_p) \subseteq \gamma_M(s_1') \times \cdots \times \gamma_M(s_p') \).
Finally, by definition of \( \gamma_M \), we can conclude that \( \gamma_M(\text{St}) \subseteq \gamma_M(\text{St}') \).

(Galois). Let \( y \in p(Stack(\text{Values})) \) and let \( \text{St} \in y \).
By monotonicity of \( \gamma_M \), we have that \( \{y \mid \gamma_M(y) \subseteq \gamma_M(y) \} \subseteq \gamma_M(\alpha_M(y)) \).
Using monotonicity of \( \gamma_M \), this entails \( \gamma_M(\alpha_M(\{y\})) \subseteq \gamma_M(\alpha_M(y)) \).
If \( \text{St} = () \) then \( \gamma_M(\alpha_M(\{\}) = () \) and thus, by the previous inclusion, \( \text{St} = () \in \gamma_M(\alpha_M(y)) \).
If \( \text{St} = s_1, s_2, \ldots, s_p, \) for some \( p > 0 \), then, by definition, \( \alpha_M(\{\}) = \alpha_A(\{s_1\}), \alpha_A(\{s_2\}), \ldots, \alpha_A(\{s_p\}) \).
And, also by definition, \( \gamma_M(\alpha_M(\{\})) \) is the set of all stacks formed by the tuples in \( \gamma_A(\alpha_A(\{s_1\}) \times \cdots \times \alpha_M(\{s_p\}) \).
By the fact that \( (\alpha_A, \gamma_A) \) is a Galois Insertion we can say that the tuple \( (s_1, s_2, \ldots, s_p) \) is an element of this set. This implies \( \text{St} \in \gamma_M(\alpha_M(\{\})) \) and, by the previously proved inclusion, \( \text{St} \in \gamma_M(\alpha_M(y)) \).

(Insertion). Let \( \text{St} \in Ab^\gamma(A) \). The thesis is trivial if \( \text{St} = \perp_M \) or \( \text{St} = \top_M \) or \( \text{St} = () \). Suppose that \( \text{St} = s_1, s_2, \ldots, s_p, \) for some \( p > 0 \). Using the definitions, \( \alpha_M(\gamma_M(\alpha_M(\{s_1,s_2,\ldots,s_p\})) = \alpha_A(\gamma_M(s_1)), \alpha_A(\gamma_M(s_2)), \ldots, \alpha_A(\gamma_M(s_p)) \).
We have that \( \forall j \in \{1, 2, \ldots, p\}, \alpha_M(s_j) = s_j \) because \( (\alpha_A, \gamma_A) \) is a Galois Insertion. Thus, \( \text{St} \in \alpha_M(\gamma_M(\text{St})) \).
Proof of Lemma 1.

Proof. For property (i) it is sufficient to observe that the inconsistency of a set of values in $y$, at $\text{ret}$ instructions, that are not consistent with some other values at calling points (recall the definition of the predicate and the examples above). If $y \subseteq y'$, then the same values are also in $y'$ and, thus, this set is inconsistent.

For property (ii), observe that the operator $\text{Ret}_{\text{Conc}}_D$ subtracts from $\gamma_{(M, St)}(S_i)$, where $i$ is such that $P(i) = \text{ret}$, the values that are inconsistent w.r.t. those in all the sets $\gamma_{(M, St)}(S)_{i-1}$ such that $P(\ell) = \text{jas} L$. By monotonicity of $\gamma_{(M, St)}$ and by the definition of $\models_{\text{AbDom}}(D)$, we know that $\forall i \in \{0, \ldots, K-1\}, \gamma_{(M, St)}(S_i) \subseteq \gamma_{(M, St)}(S'_{i-1})$. We prove now the set inclusion. Suppose that $S_i = (M_i, St_i)$, $S'_{i-1} = (M'_{i-1}, St'_{i-1})$ and $\text{SubStart}(i) = L$. Let $\ell \in \text{StaticReturnPoints}(L)$ and let $(M, St) \in \gamma_{(M, St)}(S_i)$ such that $M(x) = (i, \text{ret}(L))$. Suppose also that $\forall x' \in \{1, \ldots, N\} \cap \text{MR}(L), M(x') \in \gamma_D(M_{i-1}(x') \cap \forall x' \in \text{MR}(L), M(x') = (\omega, \Omega) \Rightarrow (\omega, \Omega) \in \gamma_D(M_{i-1}(x'))$, where $\gamma_{(M, St_{i-1})} = \gamma_{(M, St)}$. By definition, it holds that $s = (i, (M, St)) \in \text{Ret}_{\text{Conc}}_D(i, L, S, x)$. By monotonicity, $\gamma_{(M, St)}(S_i) \subseteq \gamma_{(M, St)}(S'_{i-1})$ and, thus, $(M, St) \in \gamma_{(M, St)}(S'_{i-1})$. By the properties of $\gamma_{(M, St)}$ and from the structure of the abstract domain we can say that, if $S'_{i-1} = (M'_{i-1}, St'_{i-1})$, then $\forall x' \in \{1, \ldots, N\}, \gamma_D(M_{i-1}(x')) \subseteq \gamma_D(M'_{i-1}(x'))$. Thus, it also holds that $\forall x' \in \{1, \ldots, N\} \cap \text{MR}(L), M(x') \in \gamma_D(M_{i-1}(x')) \cap \forall x' \in \text{MR}(L), M(x') = (\omega, \Omega) \Rightarrow (\omega, \Omega) \in \gamma_D(M_{i-1}(x'))$. Putting it together, we get that $s = (i, (M, St)) \in \text{Ret}_{\text{Conc}}_D(i, L, S', x)$. □

Proof of Proposition 5.

Proof. ($\alpha$-Monotonicity) Let $y, y' \in \psi(\text{States})$ such that $y \subseteq y'$. If $\neg \text{Cons}_D(y)$, by the property (i) of Lemma 1, also $\neg \text{Cons}_D(y')$. Thus, in this case, $\alpha_{\text{AbDom}}(D)(y) = \alpha_{\text{AbDom}}(D)(y') = \gamma_{\text{AbDom}}(D)$ and the thesis is trivially true. Suppose, now, that $\gamma_{\text{AbDom}}(D)$ is true and let $S = \gamma_{\text{AbDom}}(D)(y)$. If $\neg \text{Cons}_D(y)$, then $\alpha_{\text{AbDom}}(D)(y') = \gamma_{\text{AbDom}}(D)$ and, trivially again, $S \models_{\text{AbDom}}(D) \gamma_{\text{AbDom}}(D)$. If, instead, also $\gamma_{\text{AbDom}}(D)$ is true, then let $S' = \alpha_{\text{AbDom}}(D)(y')$ and let $i \in \{0, \ldots, K-1\}$. By the definition of $\gamma_{(M, St_p)}$ and by the definition of $\alpha_{\text{AbDom}}(D)$, we have to prove that $S_i = \alpha_{(M, St_p)}((M, St) \mid (i, (M, St)) \in y) \models_{(M, St_p)} S'_i = \alpha_{(M, St_p)}((M, St) \mid (i, (M, St)) \in y')$. This simply follows from monotonicity of $\alpha_{(M, St_p)}$ and from $y \subseteq y'$.

($\gamma$-Monotonicity) It follows directly from the definition of $\gamma_{\text{AbDom}}(D)$ and from property (ii) of Lemma 1.

(Galois) Let $y \in \psi(\text{States})$. If $\neg \text{Cons}_D(y)$, then $\alpha_{\text{AbDom}}(D)(y) = \gamma_{\text{AbDom}}(D)$ and, by definition of $\gamma_{\text{AbDom}}(D)$, we have that $\gamma_{\text{AbDom}}(D)(\alpha_{\text{AbDom}}(D)(y)) = \gamma_{\text{AbDom}}(D)$. Thus, in this case, the thesis is trivially true. If, instead, $\text{Cons}_D(y)$ is true, define, for all $i \in \{0, 1, \ldots, K-1\}$ the set $y_i = \{(M, St) \mid (i, (M, St)) \in y\}$. Let $S = \alpha_{\text{AbDom}}(D)(y) = \{(M, St)(y_0), \ldots, \alpha_{\text{AbDom}}(D)(y_{K-1})\}$, by definition. Let $i \in \{0, 1, \ldots, K-1\}$. Suppose that $P(i) \neq \text{ret}$. We know, from Proposition 4 applied to the domain $D$, that $\alpha_{(M, St_p)}$ and $\gamma_{(M, St_p)}$ form a Galois insertion. Thus, we have that $y_i \subseteq \gamma_{(M, St)}(\alpha_{(M, St_p)}(y_i))$. Applying the definition of $\gamma_{\text{AbDom}}(D)$, we get that $\{i, \sigma \mid \sigma \in y_i\} \subseteq \gamma_{\text{AbDom}}(D)(\alpha_{\text{AbDom}}(D)(y))$. Suppose, now, that $P(i) = \text{ret} \wedge \text{SubStart}(i) = L$. We are still supposing that $\text{Cons}_D(y)$ is true, thus the three conditions defining $\text{Cons}_D$ (recall Figure 14) are met by $y$, i.e., concrete return addresses are correct w.r.t. the subroutine structure and values in unmodified registers at return points are consistent with those at calling points. This entails that, letting $S = \gamma_{\text{AbDom}}(D)(y)$ above, every $(M, St) \in y_i$ satisfies the three conditions defining the set $\text{Ret}_{\text{Conc}}_D(i, L, S, x)$ (recall Figure 15) because they express one by one the same conditions of $\text{Cons}_D$, in the concretization context. Thus, again by definition of $\gamma_{\text{AbDom}}(D)$, we have that $\{i, \sigma \mid \sigma \in y_i\} \subseteq \gamma_{\text{AbDom}}(D)(\alpha_{\text{AbDom}}(D)(y))$. We can conclude that $y = \bigcup_{i=0}^{K-1} \{i, \sigma \mid \sigma \in y_i\} \subseteq \gamma_{\text{AbDom}}(D)(\alpha_{\text{AbDom}}(D)(y))$.

(Connection) Let $S = (S_0, S_1, \ldots, S_{K-1}) \in \text{AbDom}(D)$. Consider the definition of $\gamma_{\text{AbDom}}(D)$ in Figure 13. The $\text{Ret}_{\text{Conc}}_D$ operator is used to avoid the generation of some concrete states that lead to an inconsistent set of concrete states. Let $S' = (S'_0, \ldots, S'_{K-1}) = \alpha_{\text{AbDom}}(D)(\gamma_{\text{AbDom}}(D)(S))$. If $P(i) \neq \text{ret} x$ we have that $S'_i = S_i$ using the Insertion property of the Galois Insertion formed by $\alpha_{\text{AbDom}}(D)$ and $\gamma_{\text{AbDom}}(D)$. Else, it is easy to see, by definition of $\text{Ret}_{\text{Conc}}_D$, that $\{(M, St) \mid P(i) = \text{ret} x, \text{SubStart}(i) = L, (i, (M, St)) \in \text{Ret}_{\text{Conc}}_D(i, L, S, x)\} \subseteq \gamma_{\text{AbDom}}(D)(S_i)$. By monotonicity of $\alpha_{\text{AbDom}}(D)$, we have $\alpha_{\text{AbDom}}(D)(\{(M, St) \mid P(i) = \text{ret} x, (i, (M, St)) \in \text{Ret}_{\text{Conc}}_D(i, L, S, x)\}) \models_{\text{AbDom}}(D) \alpha_{\text{AbDom}}(D)(\gamma_{\text{AbDom}}(D)(S_i))$. By definition of $\gamma_{\text{AbDom}}(D)$ and $\gamma_{\text{AbDom}}(D)$ and by the Insertion property of $\alpha_{\text{AbDom}}(D)$ and $\gamma_{\text{AbDom}}(D)$, the previous relation can be written as $S'_i \models_{\text{AbDom}}(S_i)$. Thus, for all $i \in \{0, \ldots, K-1\}$, we can conclude that $S'_i \models_{\text{AbDom}}(S_i)$.
Proof of Proposition 6.

Proof. Applying definitions we have to show that

\[ \bigcup_{(\mathit{M}_0, \mathit{St}_0)} \{ S'' \mid S'' \mathrel{\xrightarrow{\text{next}}}_{\mathit{AbDom}(\mathit{D})} S'' \} \}

supposing \( S \mathrel{\xrightarrow{\text{next}}}_{\mathit{AbDom}(\mathit{D})} S' \). This is true if every rule for \( \mathit{AbDom}(\mathit{D}) \) (Figure 17 and the error rules) is monotone, that is to say, if the updated state of each rule grows as the state in which the rule is executed grows. Thus, suppose \( S \subseteq \mathit{AbDom}(\mathit{D}) \). We show that performing the same abstract step on \( S \) and \( S' \), i.e. selecting an \( i \) such that \( \mathcal{S}_i \neq \mathit{AbDom}(\mathit{D}) \), we get two new states \( S'' \) (from \( \mathit{S} \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \)) and \( S'' \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \). First note that if \( S \) fires an error rule then so does \( S' \). This is because all the unsatisfied premises on \( \mathcal{S}_i \), when \( \mathcal{S}_i \neq \mathit{AbDom}(\mathit{D}) \), are satisfied after an abstract state \( \mathcal{S}_i' \) such that \( \mathcal{S}_i \subseteq \mathit{AbDom}(\mathit{D}) \). In this case \( S'' \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \) and another trivial case occurs when \( S \) does not fire an error rule while \( S' \) does: \( S'' \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \). When neither \( S \) nor \( S' \) fires an error rule, we have in all cases that \( S'' \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \). The verification that \( \mathit{modified}(S_i) \subseteq \mathit{AbDom}(\mathit{D}) \) can be done for each rule. For brevity we show this verification only for some rules. Let \( S_i = (M_i, \mathit{St}_i) \), \( S_i' = (M_i', \mathit{St}_i') \) and so on for all involved abstract JVM states.

\( \text{iconst} \# \) \( \text{modified}(S_i) = \langle M_i, \text{int-St}_i \rangle, \) \( \text{modified}(S_i') = \langle M_i', \text{int-St}_i' \rangle \) because \( M_i \mathbin{\triangleleft} M_i' \) and \( \mathit{St}_i \mathbin{\triangleleft} \mathit{St}_i' \) by hypothesis.

\( \text{istore} \# \) \( \text{modified}(S_i) = \langle M_i[^{\text{int}}/x], \mathit{pop}(\mathit{St}_i) \rangle, \) \( \text{modified}(S_i') = \langle M_i'^{\text{int}}/x, \mathit{pop}(\mathit{St}_i') \rangle \)

\( \text{ret} \# \) We have to verify that given \( \ell \in \text{Static-Return-Points}(L) \), \( (S_{l-1} \triangleright_{L} (M_i[^{\text{top}}/x], \mathit{St}_i)) \subseteq (M_i[^{\text{top}}/x], \mathit{St}_i) \) by the hypothesis and by definition of \( \triangleright_{L} \), this is true if \( \forall x' \in \mathit{MR}(L) \), \( (M_{l-1} \triangleright_{L} M_i[^{\text{top}}/x](x')) \subseteq D \) and \( (M_{l-1} \triangleright_{L} M_i[^{\text{top}}/x]) \). The latter condition follows easily from \( S \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\mathit{D}) \). We have checked exhaustively all the possibilities.

Proof of Proposition 7.

Proof. Let \( \gamma \in \mathcal{P}(\mathit{States}) \). First note the cases in which the thesis is trivially true; i.e. when \( \gamma = \emptyset \) or \( \mathit{ConsD}(\gamma) = \text{false} \). In the former case it is sufficient to observe that \( \text{next}^{C}(\emptyset) = \emptyset \). For the latter case, observe that \( \alpha_{\mathit{AbDom}(\gamma)}(\gamma) = \top_{\mathit{AbDom}(\gamma)} \) and that \( \text{next}^{A}(\gamma) = \top_{\mathit{AbDom}(\gamma)} \). When \( \gamma \) is consistent and not empty, we can partition it w.r.t. the program points. Let \( \text{Select}(i, y) = \{(i, (M, \mathit{St})) \in \mathit{States} \mid (i, (M, \mathit{St})) \in y\} \). This means that \( \gamma \mathrel{\xrightarrow{\text{next}}}_{\mathit{next}} \mathit{AbDom}(\gamma) \). Let \( \text{Succ}(i) \) be the set of all successors program points of the instruction \( \gamma \), i.e. when \( \mathcal{S}_i \neq \mathit{AbDom}(\gamma) \). We can partition also the set \( \text{next}^{C}(\gamma) \) by indexes considering the parts \( \text{next}^{C}(\gamma, i) \) of \( \gamma \). We show that each of these parts is a subset of \( \gamma_{\mathit{AbDom}(\gamma)}(\gamma) \). Let \( \text{Succ}(i) \) be the set of all successors program points of the instruction \( i \), i.e. when \( \mathcal{S}_i \neq \mathit{AbDom}(\gamma) \). We can partition also the set \( \text{next}^{C}(\gamma, i) \) by indexes considering the parts \( \text{next}^{C}(\gamma, i) \) of \( \gamma \). We show that each of these parts is a subset of \( \gamma_{\mathit{AbDom}(\gamma)}(\gamma) \).

\( 14 \) See the proof of the previous proposition.
\{M'' \mid (i, (M'', St'')) \in y\} \Rightarrow M'' \in St_i \Rightarrow M_i \Rightarrow \gamma_{M_D}(\alpha_{M_D}(\{M'' \mid (i, (M'', St'')) \in y\}) = M_i \Rightarrow \gamma_{M_D}(\alpha_{M_D}(\{M'' \mid (i, (M'', St'')) \in y\}) \subseteq \gamma_{M_D}(M_i).

Thus, \(\gamma_D(M_i(x')) \subseteq \gamma_D(M_{i-1}(x'))\) and, using the property (Galois) of the Galois insertion, \(M(x') \in \gamma_D(M_i(x')) \subseteq \gamma_D(M_{i-1}(x'))\).

\[\Box\]

### A.2. DETECTING SUBROUTINES BOUNDARIES

In this Appendix we show a labelling procedure which can be used to determine MR. The procedure is essentially the one defined in [27]. Here we simply define it as an abstract interpreter and we show an example of labelling.

We start defining a sequence of subroutine calls, which is a sequence of addresses of instructions which call a subroutine. As in [27] we assume that each subroutine has a single entry point reachable by a jsr instruction, and that each entry point contains an astore instruction. Moreover, we assume that astore instructions at different entry points act on different registers. This guarantees that a register, containing a return address, is associated to a single subroutine. Finally, we do not allow recursive subroutine calls.

**Definition A.1 (Sequence of subroutine calls).** A sequence of subroutine calls is a sequence of addresses of calling instructions \(i_1 \circ i_2 \circ \ldots \circ i_n\), built by the concatenation operator \(\circ\). The empty sequence is denoted by \(\epsilon\). For each address, \(i\), in a sequence of subroutine calls, a return register, RetReg(i), and a subroutine first address, FirstAddress(i), are defined. Namely, if \(P(i) = \text{jsr } L\) and \(P(L) = \text{ astore } x\) then RetReg(i) = x and FirstAddress(i) = L.

Figure A.1 shows the rules of the abstract interpreter. Every instruction \(i\) of the given bytecode program is associated with \(S^i_F\), which is either \(\bot\) or a set of sequences of subroutine calls.

We associate, to each instruction, a set of addresses of subroutine calls instead of a single address. This allows to handle sequences of instructions, belonging to different subroutines, accessed by jumps. In this case, the sequences of subroutine calls relative to different subroutines belong to the set.

The initial state \(S^i_L\) is such that \(S^i_L = \{\epsilon\}\) and \(S^i_L = \bot\) for all \(i = 1, 2, \ldots, K - 1\).

Every rule but those for jsr and ret merges (by the \(\cup_L\) operator) the set of sequences with the ones of the successors instructions. The rules for jsr and ret handle such a set as follows. When a subroutine is called at a program point \(i\) (rule jsr\(^L\)), the function SubCall adds to each sequence in \(S^i_L\) (the subroutine call sequences at the entry point of the subroutine) the address of the subroutine call. When a subroutine returns, by a ret \(x\) instruction, its

\[\text{This assumption is needed for handling the cases in which there are instructions belonging to different subroutines, e.g. in nested subroutines.}\]
We can univocally determine the returning subroutine, from the subroutine call sequences. Note that, for call address is removed, by the function `SubReturn`, because only one address, \( i \), in the sequence, is such that \( \text{RetReg}(i) = x \).

We allow to handle multilevel returns: when a subroutine returns, the function `SubReturn` removes from the subroutine call sequences the addresses of all

### FIGURE A.1

The rules of the labelling interpreter

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Condition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iconst</code></td>
<td>( P(i) = \text{iconst } c ), ( S^t_i \neq \bot )</td>
<td>( i + 1 \in {0, 1, \ldots, K-1 } )</td>
<td>( S^t \rightarrow_{\text{next}} S^t[S^t_{i+1} := S^t_{i+1} \cup L S^t_i] )</td>
</tr>
<tr>
<td><code>iload</code></td>
<td>( P(i) = \text{iload } x ), ( S^t_i \neq \bot )</td>
<td>( x \in {1, 2, \ldots, N} )</td>
<td>( i + 1 \in {0, 1, \ldots, K-1 } )</td>
</tr>
<tr>
<td><code>istore</code></td>
<td>( P(i) = \text{istore } x ), ( S^t_i \neq \bot )</td>
<td>( x \in {1, 2, \ldots, N} )</td>
<td>( i + 1 \in {0, 1, \ldots, K-1 } )</td>
</tr>
<tr>
<td><code>astore</code></td>
<td>( P(i) = \text{astore } x ), ( S^t_i \neq \bot )</td>
<td>( x \in {1, 2, \ldots, N} )</td>
<td>( i + 1 \in {0, 1, \ldots, K-1 } )</td>
</tr>
<tr>
<td><code>ifeq</code></td>
<td>( P(i) = \text{ifeq } L ), ( S^t_i \neq \bot )</td>
<td>( i + 1, L \in {0, 1, \ldots, K-1 } )</td>
<td>( S^t \rightarrow_{\text{next}} S^t[S^t_L := S^t_L \cup L S^t_i, S^t_{i+1} := S^t_{i+1} \cup L S^t_i] )</td>
</tr>
<tr>
<td><code>goto</code></td>
<td>( P(i) = \text{goto } L ), ( S^t_i \neq \bot )</td>
<td>( L \in {0, 1, \ldots, K-1 } )</td>
<td>( S^t \rightarrow_{\text{next}} S^t[S^t_L := S^t_L \cup L S^t_i] )</td>
</tr>
<tr>
<td><code>jsr</code></td>
<td>( P(i) = \text{jsr } L ), ( P(L) = \text{astore } x ), ( S^t_i \neq \bot )</td>
<td>( x \in {1, 2, \ldots, N} ), ( L \in {0, 1, \ldots, K-1 } )</td>
<td>( S^t \rightarrow_{\text{next}} S[S^t_L := S^t_L \cup L \text{SubCall}(i, S^t_i)] )</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>( P(i) = \text{ret } x ), ( S^t_i \neq \bot )</td>
<td>( x \in {1, 2, \ldots, N} ), ( {(l_1, C_{s_1}), (l_2, C_{s_2}), \ldots, (l_p, C_{s_p})} = \text{SubReturn}(x, S^t_i) )</td>
<td>( S^t_L := S^t_L \cup L C_{s_1}, S^t_{l_2} := S^t_{l_2} \cup L C_{s_2}, \ldots, S^t_{l_p} := S^t_{l_p} \cup L C_{s_p} )</td>
</tr>
<tr>
<td><code>ireturn</code></td>
<td>( P(i) = \text{ireturn } )</td>
<td>( S^t_i \neq \bot )</td>
<td>( S^t \rightarrow_{\text{next}} S^t )</td>
</tr>
</tbody>
</table>

### FIGURE A.2

Definition of functions `SubCall` and `SubReturn`

\[
\text{SubCall}(i, S^t_i) = \{ i \circ CS \mid CS \in S^t_i \}
\]

\[
\text{SubReturn}(x, S^t_i) = \{ (l, C_{s_l}) \mid CS' \circ l \circ C_{s_l} \in S^t_i \}
\]

where \( P(l) = \text{jsr } L \)

and \( P(L) = \text{astore } x \)
Barbuti et al.

\[
\begin{align*}
\bot_L \cup_L S_i^L &= S_i^L \\
S_i^L \cup_L S_j^L &= \{ Cs | Cs \in S_i^L \cup_L S_j^L \\
&\quad \land (\exists Cs', Cs'' : Cs = Cs' \circ Cs'' \land Cs'' \in S_i^L \cup_L S_j^L) \}
\end{align*}
\]

**FIGURE A.3.** Definition of \( \cup_L \)

```
static void m(boolean b) {
    goto E  // \{ e \}
}
while(x) {
    A: iconst 0  // \{ e \}
    try {
        istore 1  // \{ e \}
        x = false;  // \{ e \}
        goto E     // \{ e \}
    } finally {
        if (x) continue;  // \{ e \}
    }
    B: jsr C  // \{ e \}
    istore 1    // \{ B \}
    iload 1     // \{ B \}
    ifeq D       // \{ B \}
    goto E       // \{ B \}
    C: astore 2   // \{ e \}
    goto E       // \{ e \}
    D: ret 2      // \{ e \}
    goto E       // \{ B \}
    E: iload 1    // \{ B \}
    ifne A        // \{ e \}
    return       // \{ e \}
}
```

**FIGURE A.4.** An example of an exit from a subroutine without a `ret` instruction

the subroutines called while the returning subroutine was active.

It is important to remark that removing the restriction that a register, containing a return address, is associated to a single subroutine could cause to find two, or more addresses, \( i_1, i_2, \ldots i_k \), in a subroutine call sequence, such that \( \text{RetReg}(i_1) = \text{RetReg}(i_2) = \ldots = \text{RetReg}(i_k) = x \). In such a case it is not possible, for the function \( \text{SubReturn} \), to determine the returning subroutine. Wildmoser, in [27], considers such subroutines as having malformed return points and rejects them.

The definitions of \( \text{SubCall} \) and \( \text{SubReturn} \) are given in Figure A.2.

A classical problem with subroutine labelling is to delimit the boundaries of subroutines. Ordinary jump instructions may be used to terminate subroutines, thus it is difficult to say whether an instruction belongs to a subroutine or not.

Here we adopt a fixpoint version of the solution presented in [27]. A jump from an address \( i \) to an address \( j \) associated with a subroutine call sequence which is a prefix of the one at \( i \) is considered as an exit from a subroutine. This is accomplished by the \( \cup_L \) operator on subroutine call sequences, as defined in Figure A.3, which merges subroutine call sequences by computing their common prefixes.

When a jump reaches a program address \( j \) that has already been reached under a shorter sequence of subroutine calls, we know that \( j \) belongs to an outer subroutine. This situation corresponds to a return, without a `ret` instruction, from an inner subroutine. This return is recorded by deleting, at \( j \), the call sequences of the inner subroutine and by retaining the call sequences of the outer one: that is the shortest of the sequences with overlapping prefixes. An example of this technique can be found in Figure A.4, at address \( E \), where the subroutine starting at \( C \) returns with a `goto` instruction.

The defined labelling procedure produces a set of subroutine calls \( S_i^L \), for each instruction address \( i \). For each sequence \( i_1 \circ \ldots \circ i_n \in S_i^L \), \( \text{FirstAddress}(i_1), \ldots, \text{FirstAddress}(i_n) \) are the starting points of all the subroutines to which \( i \) belongs.

From the sets of subroutine call sequences we can compute \( \mathfrak{R}(L) \) as we need to define our abstract interpretation. In particular, for each instruction \( i \), we can determine, by using the function \( \text{FirstAddress} \), the starting point of all the subroutines to which \( i \) belongs.

The labelling procedure is able to correctly label the program in Figure A.4, taken from [8], which has an exit
from a subroutine without a `ret` instruction deriving from the compilation of a `continue` Java construct inside a `finally` inside a `while`.

REFERENCES


