A Practical Mobile-Code Format with Linear Verification Effort

Ning Wang and Michael Franz

Technical Report 03-26
School of Information and Computer Science
University of California, Irvine, CA 92697-3425

Nov. 18, 2003

Abstract

We present an abstract machine that encodes both type safety and control safety in an efficient manner and that is suitable as a mobile-code format. At the code consumer, a single linear-complexity algorithm performs not only verification, but simultaneously also transforms the stack-based wire format into a register-based internal format. The latter is beneficial for interpretation and native code generation. Our dual-representation approach overcomes some of the disadvantages of existing mobile-code representations, such as the JVM and CLR wire formats.

Contents

List of Figures

List of Tables

1 Introduction

Mobile programs can potentially be malicious. To protect itself, a host that receives such mobile programs from an untrusted party or via an untrusted network connection will want some kind of guarantee that the mobile code is not about to cause any damage. The Java Virtual Machine (JVM) pioneered the concept of *code verification* by which a receiving host examines each arriving mobile program to rule out potentially malicious behavior even before starting execution.

Unfortunately, verification in itself consumes computing resources and thereby represents an overhead. Moreover, the need for eventual verification at the code recipient prevents many traditional compiler optimizations at the code producer. As a result, first-generation verifiable mobile code formats such as JVM code and the ECMA CLR intermediate language require substantial just-in-time compilation effort at the code recipient to obtain reasonable execution performance.

The very concept of verification also presents a potential avenue for an attack. Gal et al. [7] have recently shown how a carefully crafted mobile program in the JVM bytecode language (JVML) can result in a denial of service attack when sent as an "applet" to a client computer or as an "agent" to a server. Because worst-case verification complexity in JVML does not grow linearly with method size, one can craft relatively short programs (a few thousand bytes) that require an extreme verification effort (on the order of hours on a workstation-class machine) before they are finally recognized as valid.

The main characteristic of what we call "first-generation" mobile code representations (e.g., JVML and the ECMA CLR intermediate language) is that they are based on virtual machines with *unrestricted goto instructions* and *untyped expression stacks*. Both of these features represent major hurdles to (a) producer-side code optimizations, (b) to verification efficiency, and (c) to high-quality just-in-time code generation.

In the meantime, alternative schemes for mobile-code distribution have appeared that differ from the model exemplified by the JVM and the CLR. Proof-carrying code (PCC) approaches [14, 15] involve the code producer in the verification process by obliging it to construct a proof of safety. The code consumer computes a verification condition from the received program and checks whether the proof supplied by the producer discharges the computed verification condition. PCC requires only linear (in the length of the proof) effort on the code consumer. An early concern that the proofs were often larger than the programs themselves has been put to rest by proof compaction techniques [16].

Amme et al. [3] have proposed a mobile-code format based on Static Single

Assignment Form that is both type-safe and control-safe. This format, SafeTSA, retains the high-level control structures of the original source language, such as *while*, *if*, and *for loops*. As a consequence, when the origin of a mobile program is not source code, but existing code in another representation providing unrestricted gotos (for example, JVML), this becomes a problem. Increasingly, JVM code is run through code obfuscators prior to distribution, creating irreducible control flow. Such code cannot easily be mapped back onto high-level control structures, diminishing the applicability of representations such as SafeTSA.

Moreover, SSA-based representations are meant to be compiled on-the-fly rather than interpreted. A recent attempt at interpreting SSA directly [23] has resulted in success, but at the expected low performance point. Hence, for the foreseeable future, SSA-based mobile code formats are likely to remain restricted to workstation-class devices with large memories and ample processor resources.

In this paper, we present a new approach to transmitting mobile code that is based on an abstract machine that has two parts, a *wire format* emitted by the code producer and transmitted to the code consumer, and an *internal format* seen only by the code consumer. The internal format is generated during verification of the wire format, and the effor for the verification/translation step is linear in the length of the wire representation.

Our abstract machine, which we call the *Certificate Abstract Machine (CAM*), is able to completely capture all control flow that might result from running JVM bytecode through a code obfuscator. Hence, we can translate directly from JVML or CLR code into our wire format. Instead of providing difficult-to-verify unrestricted gotos, our wire format provides complete inherent control-structure safety. As we will explain below, the key to our representation is that it encodes the *dominance* relationship and *variable scoping* directly.

The internal format of our abstract machine is register based and can be interpreted efficiently on resource-constrained devices. Alternatively, it can also be translated quickly just-in-time into high-quality native code where the appropriate processor resources are available. Hence, it covers the complete applicability range of the existing JVM and CLR formats at a better performance point.

In the following, we first present key terminology. We then present the key idea of the CAM, namely the use of the dominator relationship for modeling control safety and scoping to model data flow and type safety. Following this brief overview, we present CAM in detail and specify the operational semantics for the CAM abstract interpreter. After discussing the current implementation and related work, we conclude our paper.

2 Preliminaries

2.1 Terminology and Notation

A *control flow graph (CFG)* $G(V, E_G)$ is a directed graph in which V is the set of nodes representing basic blocks, and an edge $(x, y) \in E_G$ represents a possible flow of control from x to y. There are two distinguished nodes: $start$ and end. $start$ has no predecessor and every node is reachable from it. end has no successors and is reachable from every node. The presence of an edge $(start, end) \in E_G$ indicates that the surrounding program might not execute G at all.

A *path* in G is a sequence of nodes $(n_1, n_2, ..., n_k)$ such that all nodes are distinct and $(n_i, n_{i+1}) \in E_G$, $1 \leq i < k$. For nodes x and $y \in V$, if x appears on every path from $start$ to y, then x *dominates* y. Every node dominates itself. Node x *strictly dominates node* y if x dominates y, and $x \neq y$. We write $x \prec y$ for strictly dominating and $x \preceq y$ for dominating. Node x is the *immediate dominator* of node y, denoted $idom(y)$, if x is the closest strict dominator of y on any path from $start$ to y. Every node, except $start$, has a unique immediate dominator. Every node has no more than one immediate dominator. The edges $E_T = \{(idom(x), x) | x \in V - \{start\}\}$ form a directed tree rooted at $start$, called the *dominator tree* denoted by $T(V, E_T)$ of $G(V, E_G)$, such that x dominates y if and only if x is an ancestor of y in the dominator tree.

2.2 Safety Requirements

In order to guarantee the abstractions of the target machine, a mobile program must satisfy all of the following:

1. Type safety – every value is used only in ways that are consistent with its type. For example, a floating point value could not be used as array index.

2. Control flow safety – control transfer instructions, i.e., branches that cause the flow of control to leave a basic block, must lead to *valid* targets. For example, procedures can only be reached at their entry point.

3. Data flow safety – every variable must be declared and initialized before it is used.

In the following, we will call a program "well-typed" when it satisfies these conditions. For programs written in high-level programming languages, a com-

piler can easily check that these criteria are met. But now we are interested in lower-level formats that *still* allow to verify these properties.

2.3 Verification

The JVM bytecode verifier checks the "well-typedness" criteria presented in Section 2.2 above. In particular, control flow safety and data flow safety are interwoven in the standard JVML verification algorithm. Verification is similar to abstract interpretation of the program, considering only the types of variables rather than their concrete values. At join points in the control flow, the verifier needs to confirm type consistency in the data flow.

In the case of Java, which provides subtyping, instructions encountered during the abstract interpretation may produce new variables or variables with new types, which in turn may affect the availability and type of other variables reachable via control flow. The updated availability and types of variables may further affect the behavior of already traversed instructions.

As a consequence of this mutual dependency between control flow and data flow (Figure 1), the JVM bytecode verifier may need to iterate until it reaches a stable state in which no new variable or new type is produced. This fixed-point iteration algorithm [17] has a worst-case performance that is quadratic. In a recent paper, Gal et al. [7] have shown how one can systematically construct JVML programs that exhibit worst-case verification behavior and use these programs in a denial-of-service attack on the machine hosting the JVM.

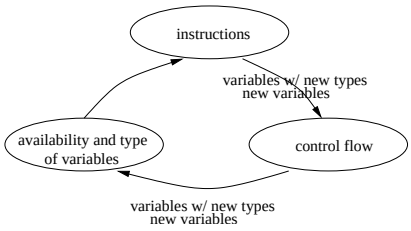


Figure 1: Dependence among instructions, variables and control flow in JVML.

In the Certificate Abstract Machine, we successfully break up the cycle linking control flow and data flow by introducing the dominator relationship directly into the mobile-code representation (Figure 2). In our verification algorithm, traversing instructions affects the availability and type of variables via dominance instead of via control flow. The updated availability and types of variables affect only the

still-unchecked instructions in type-check sequence—there is no longer a cycle. This will be explained in detail in Section 3.4 below. In Section 3.5, we will also give a definition of control safety based on dominance, and an algorithm to verify the validity of control flow.

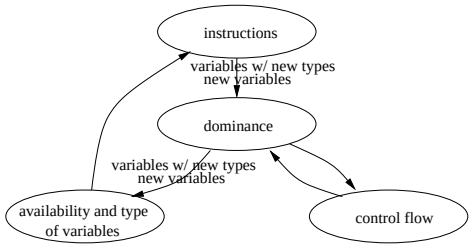


Figure 2: Dependence among instructions, variables, control flow and dominance in CAM

After these preliminaries, we can now introduce the Certificate Abstract Machine in detail.

3 Certificate Abstract Machine

Our goal is to make checking the safety criteria listed in Section 2.2 as efficient and simple as possible. The Certificate Abstract Machine is our vehicle to achieve this goal. In particular, CAM provides more efficient verification than the Java Virtual Machine, and also simpler and faster interpretation or just-in-time code generation.

The CAM uses two different representations for every program, and translates between these formats as a side-effect of verification. The *wire format* (or CAM code) is a stack-based intermediate representation that is used to transport mobile programs from one place to another. It is the representation that can be verified. We can actually generate this format directly from JVML, i.e., we do not require the presence of any Java source code, and our JVML-to-CAM compiler can even deal with bytecode that has been deliberately obfuscated.

On the code consumer side, a CAM implementation takes the CAM code and translates it into a register-based *internal format*. This translation occurs as a side-effect of verification; the internal format is never seen on the outside of the abstract machine and is executed only if verification is successful.

The verification/translation process involves a symbolic execution of the CAM code considering only the types of variables, rather than their values. Unlike

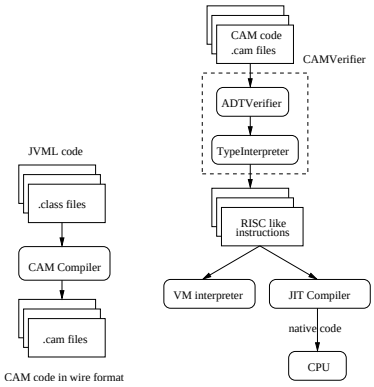


Figure 3: Life cycle—(a) translate JVML into CAM code at code producer; (b) verify and execute CAM code at code consumer

JVML verification, this process occurs in linear time. During this type-based execution, types are loaded from the constant pool or register file onto a type expression stack and operations are executed on these types. If the CAMVerifier successfully completes execution, then the CAM code is safe and the register-based internal format is emmitted to the VM Interpreter or JIT compiler. The lifecycle of CAM from producer to consumer is illustrated in Figure 3. The rest of this section will first present an example, and then introduce the various components of CAM in more detail.

3.1 An Example

The best way to illustrate the differences between JVML code and CAM code is by way of an example. The Java source program in Figure 4(a) computes $n!$ if n mod 2 is zero, otherwise it computes 2^n. The variable x is declared as a different type in the different cases. The syntax of Java successfully assigns different scopes to $x : int$ and $x : long$; hence, there is not conflict between them.

Figure 4(b) shows the JVML code corresponding to this example and Figure 5(a) the CAM code. The major differences between CAM and JVML are the following: JVML has explicit "gotos" and conditional branches with explicit jump targets. In CAM code, on the other hand, the control flow targets are encoded in an augmented dominator tree (ADT). Unlike the JVM, CAM has explicit block instructions to delimit basic blocks (operational semantics will be given below). All CAM instructions are designed to operate over types instead of values. CAM has a **RegisterFile**, a typed **ConstPool** and a **TypeOpStack**. The loadr instruction

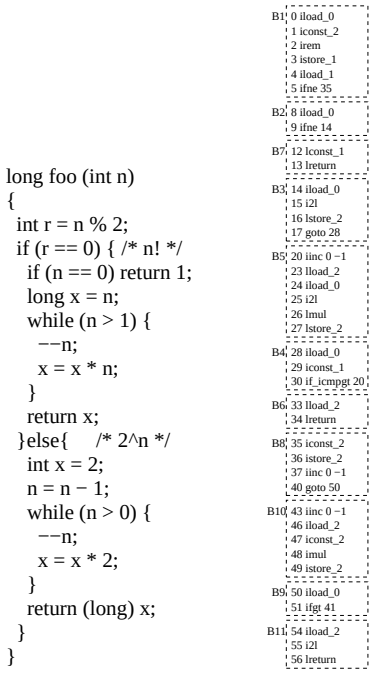


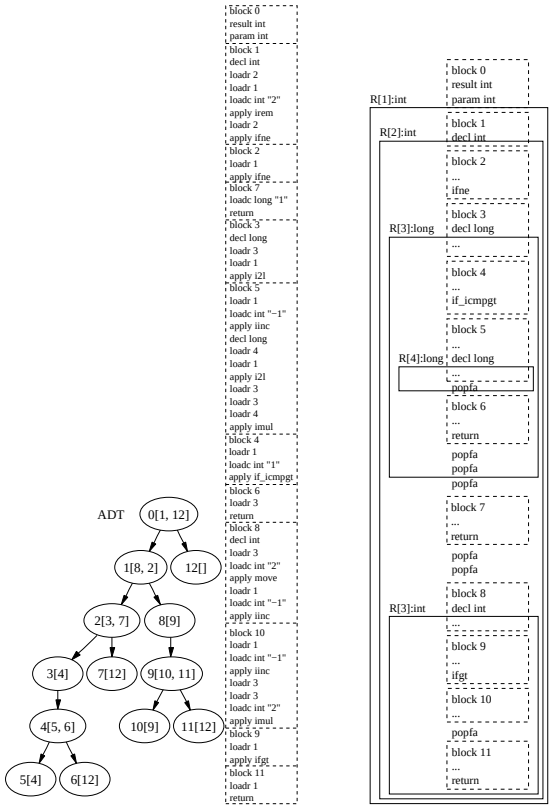
Figure 4: (a) Java source and (b) JVM bytecode for the running example

in CAM is untyped: it copies a type from the **RegisterFile** to the **TypeOpStack**. The instruction decl assigns the next available register to a type. These three and other specifi c instructions, all of which are explained further below, represent the model that we use for safety checking—they disappear after verifi cation when the code is re-written into its internal representation (Figure 6).

Finally, the most important and unique property of CAM is *scoping*. The scope of each local variable in JVML is unclear. CAM lets us recover instruction sequences that have non-overlapping scopes for each register:type pair. Figure 5(b) illustrates this point. The $x : int$ and $x : long$ in the original Java code correspond to $R[3] : int$ and $R[3] : long$ respectively.

3.2 The Verifier

The CAM Verifi er has two logically indepedent parts: a type-level abstract interpreter (TypeInterpreter) that operates on types instead of values, and an augmented dominator tree verifi er (ADTVerifi er) that makes sure that control flows



(a) (b)

Figure 5: (a) CAM code for the same example, consisting of an augmented dominator tree and a sequence of instructions and (b) code sequence executed by the verifi er (boxes indicate the scopes of the associated register:type pairs)

go only to proper targets. A program represented in CAM code is well-typed if both ADTVerifi er and TypeInterpreter return true.

A set of transition functions over types constitutes the interpretation rules of the TypeInterpreter. It interprets instructions in a sequence in which the scope of any variable v covers only the instructions following the $define$ instruction of v. We call this the **type-check sequence** as it is used only by the TypeInterpreter. The type-check sequence doesn't necessarily have to coincide with either the program sequence or control flow sequence. One example is show in Figure 5(b). The algorithm generating the type-check sequence is given below in section 3.4.4.

The TypeInterpreter completely ignores control flow instructions. It interprets

```
B1   1 rem R[2] R[1] 2
     2 ifne R[2] 12
B2   3 ifne R[1] 5
B7   4 return 1
B3   5 i2l R[3] R[1]
     6 goto 10
B5   7 iadd R[1] R[1] -1
     8 i2l R[4] R[1]
     9 imul R[3] R[3] R[4]
B4   10 if_icmpgt R[1] 1 7
B6   11 return R[3]
B8   12 mov R[3] 2
     13 iadd R[1] R[1] -1
     14 goto 17
B10  15 iadd R[1] R[1] -1
     16 imul R[3] R[3] 2
B9   17 ifgt R[1] 15
B11  18 return R[1]
```

Figure 6: Register-based internal CAM representation of our example program, generated if verification succeeds

instructions strictly following the type-check sequence and no control transfer is ever taken into account interrupting the interpretation sequence. As a consequence, the interpretation complexity is O(l), with l being the number of instructions. If the TypeInterpreter successfully interprets all instructions in the type-check sequence, then the code doesn't voilate any type rules.

The reader might have noticed a special instruction popfa in the type-check sequence listed in Figure 5(b). These instructions are inserted during the preparation of the type-check sequence and don't appear in the CAM wire format. We elaborate on their operational semantics below.

The control safety property is encoded separately from the CAM code in the Augmented Dominator Tree (ADT). The ADTVerifier (Section 3.5), which is completely separate from the TypeInterpreter, checks the validity of the ADT in $O(E + V)$ complexity, with E being the number of edges and V being the number of vertices in the tree. If the ADT is valid, then all control flow goes to proper targets.

3.3 Syntax

The stack-based intermediate representation used as the CAM's wire format is designed solely for the purpose of verification. After successful verification, all programs in this format will be rewritten into a register based internal format. Since the register-based internal format exists only in memory and is private to each implementation of the CAM, we specify only the syntax of the stack-based wire format (Figure 7).

		$i_0, ..., i_n, i, m, n \in \mathbb{N}$	
		$v_0, ..., v_n \in \mathbb{Z}$	
$\mathbf{R}$	::=	$\{0 \mapsto \top, ..., i \mapsto \top\}$	**RegisterFile**
$\mathbf{C}$	::=	$\{0 \mapsto \langle v_0, \tau_0 \rangle, ..., i \mapsto \langle v_i, \tau_i \rangle\}$	**ConstPool**
$\mathbf{B}$	::=	$\{0 \mapsto i_0, ..., n \mapsto i_n\}$	**BlockMap**
$\mathbf{E}$	::=	$\{\}$	**ExceptionMap**
τ_{basic}	::=	$\top \vert int \vert long \vert Object...$	**BasicTypes**
τ_{array}	::=	$\tau\ (i)$	**ArrayTypes**
$\mathsf{t}_{\mathsf{abstract}}$	::=	i2l|iadd|move...	**AbstractTypes**
τ	::=	$\tau_{basic} \vert \tau_{array}$	**Types**
I_{ctr}	::=	ifne|if_icmpgt|retnv	
	|	switch$\{v_0, v_1, v_2, v_3, ..., v_n\}$	**CtrlInst**
I_{other}	::=	loadr i|loadc i|decl τ|apply $\mathsf{t}_{\mathsf{abstract}}$	
	|	arraylength τ_{array}	**OtherInst**
I	::=	$I_{ctrl} \vert I_{other} \vert$block m	
	|	popfa|result τ|param τ...	**Inst**
$Kind$	::=	$normal \vert synch \vert jsr \vert ret$	**BlockKinds**
ADT	::=	$m[n_1, n_2] jsr\ ADT*$	
	|	$m[n, ...]\{normal \vert synch \vert ret\}\ ADT*$	
	|	$m[n] ret\ ADT*$	**ADT**
$BB0$	::=	block 0 result τ $\{$param $\tau\}*$	
BB	::=	block $m\ I_{other} *\ I_{ctr}?$	**BasicBlock**
P	::=	$\mathbf{R}\ \mathbf{C}\ \mathbf{B}\ ADT\ BB0\ BB*$	**Program**

Figure 7: Syntax of the CAM wire format

CAM inherits the type system of JVML and the constant pool of JVML.[1] In our syntax description, i refers to an index into the **RegisterFile** or the **ConstPool**. Implicit coercion is not allowed as in JVML. ADT is our Augmented Dominator Tree encoding of the control flow graph. Each tree node has an unique id m followed by a successor(adjacency) list $[n, ...]$ and $Kind$. $Kind$ denotes the semantics of a tree node, $normal$ denotes normal basic block, $synch$ denotes synchronized block and ret denotes the semantics of final statement in Java source language. Except for the *end* node, every node has at least one succssor. The number of successors of node b matches the CtrlInst in the basic block b. For example "ifne" has exactally two successors.

A program declaration $\mathbf{R}\ \mathbf{C}\ \mathbf{B}\ ADT\ BB0, BB*$ defines the structure of a program. $\mathbf{R}$ is a register file and all registers are uninitialized; hence, all register have the type $\top$. $\mathbf{C}$ is a typed constant pool. $\mathbf{B}$ maps block ids to the numbers of its successors. $\mathbf{ADT}$ is the Augmented Dominator Tree encoding of the control flow graph as explained above. $\mathbf{BB0}$ defines the abstract data type of the method; its counterpart in JVML is the method signature. $\mathbf{BB}*$ is the sequence of basic blocks. Each basic block consists of a sequence of instructions. Before we explain the semantics of the CAM wire format any further, we need go into the details of the design principles of CAM.

3.4 Valid Variable Analysis

3.4.1 Data flow analysis

Form data flow analysis view points, given a control flow graph $G(V, E_G)$, a reference to x at point $\mathbf{p}$ in G is *valid* if every *path* leading to $\mathbf{p}$ from $start$ contains a prior definition of x. More precisely, from the type inference view point, assume the definitions of x prior to $\mathbf{p}$ have $\vdash x : \tau_i$, $0 \leq i \leq k$ and x is refered as τ at point $\mathbf{p}$, if $\tau_0 \trianglelefteq \tau \wedge ... \tau_k \trianglelefteq \tau$ and $\tau \triangleleft \top$, then the reference of x at point $\mathbf{p}$ is valid and x has type τ. For the examples in Figures 8(a) and 8(b), both paths leading to B3 contain definitions of x, so the reference to x in B3 is valid. Conversely, the references to x in Figure 8(c) and 8(d) are invalid because at least one path from $start$ to the reference contains no definition of x. All paths leading to point $\mathbf{p}$ from $start$ need to be considered to verify references at point $\mathbf{p}$.

If x is defined in any dominator of $\mathbf{p}$ and x has the type τ which is the supertype of any types of x which are dominated by τ, then x must appears on all paths

[1] In fact, in our implementation we re-use the complete JVM class-file format and merely replace the JVM's "code" attribute with our own combination of "CAM-code" and "ADT" attributes.

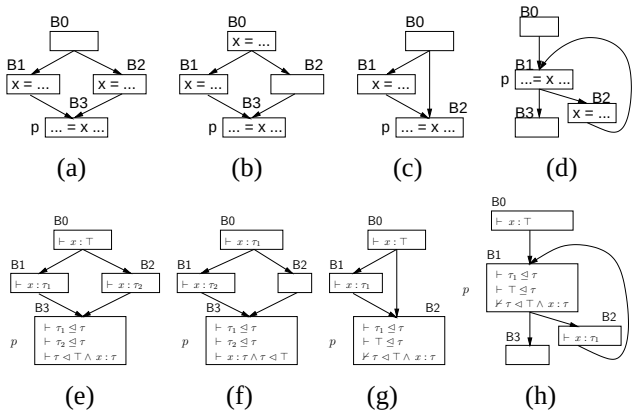


Figure 8: The references of x at point **p** are valid in (a) and (b) but are invalid in (c) and (d). (e), (f), (g), (h) show precise type conditions for valid references at point **p**.

reachable to point **p** from $start$ by the definition of the dominator relationship and if $\tau \triangle \top$, the it is valid typed. The converse is not true. Figure 9 illustrates the dominator trees of the examples in Figure 8. In the example of Figure 8(a), the reference to x at block B3 is valid, but there is no common definition of x in the dominator (B0) of B3 (see Figure 9(a)).

In this case, we can *insert a definition* of x to B0 (Figure 9(a)) without changing the behavior of the program. When applying this insertion to any program that has valid references initially, the resulting program after insertion will also be valid.

In particular, let VALID(b) be the set of valid variables on entry to block b and DEF(b) be the set of variables defined in b. We define a system of equations for VALID(b), $b \in V$:

Definition 1 (Valid variables at the entry of block b)

$$\mathbf{VALID}(b) = \begin{cases} \emptyset & b = start \\ \mathbf{VALID}(idom(b)) \cup \mathbf{DEF}(idom(b)) & b \in V - \{start\} \end{cases}$$

A variable x defined in a block b can only be referred in the dominator sub-tree rooted by b. This corresponds to the scope concept in CAM. Given the dominator

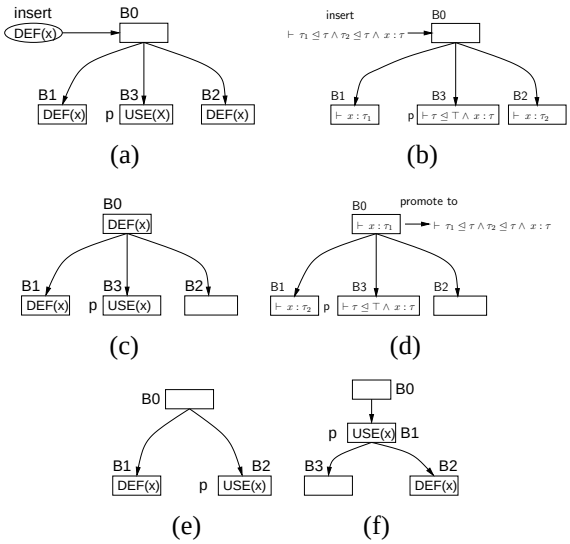


Figure 9: (a), (c), (e) and (f) are the the dominator trees of 8(a), 8(b), 8(c) and 8(d) respectively. Insertion a definition of x into B3 in (a) will not change the behavior of the program.

tree in Figure 10, the VALID set is

$$
\begin{array}{lcl}
\mathbf{VALID}(B0) & = & \emptyset \\
\mathbf{VALID}(B1) & = & \mathbf{VALID}(B0) \cup \{a : int\} \\
\mathbf{VALID}(B2) & = & \mathbf{VALID}(B1) \cup \{b : int\} \\
 & \ldots & \\
\mathbf{VALID}(B4) & = & \mathbf{VALID}(B3) \cup \{c : int\} \\
 & \ldots & \\
\mathbf{VALID}(B5) & = & \mathbf{VALID}(B4) \cup \emptyset \\
 & \ldots & \\
\mathbf{VALID}(B8) & = & \mathbf{VALID}(B1) \cup \{b : int\} \\
\mathbf{VALID}(B9) & = & \mathbf{VALID}(B8) \cup \{e : int\} \\
 & \ldots &
\end{array}
$$

The valid variable analysis of the preprocessed program in CAM is simplified vs. a program in JVML. Instead of having to iterate over all paths that lead to p from $start$, one only needs to check if there is a definition of x on the path from $start$ to point p in the dominator tree. Assume the defintion of x is referred as τ_i and $0 \leq i \leq k$, the the definition of x has the type τ ,where $\vdash \tau \lhd \top$ and $\vdash \tau_i \trianglelefteq \tau$, $0 \leq i \leq k$.

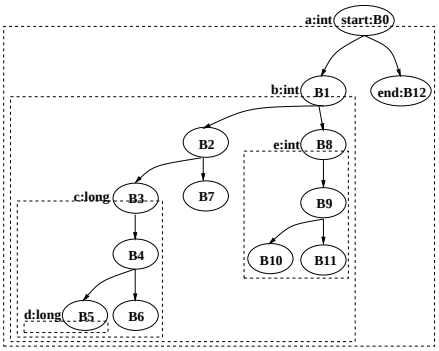


Figure 10: Dominator tree; boxes represent variable scoping

3.4.2 Type analysis

3.4.3 Temporary Registers in CAM

CAM code uses an implicit register numbering through the decl instruction. This section explains how registers are assigned internally, based on these implicit declarations. We will make use of this explanation below.

Internally, the CAM machine keeps a maximum register pointer mrp. The basic principle of register allocation then becomes: given a dominator tree. 1. We start from the root and traverse down to one leaf. 2. When a new variable definition is found, we assign $\mathbf{R}[mrp + 1]$ to that variable and increase the value of mrp by one. 3. We reset mrp to 0, and repeat steps 1 and 2 until all paths from root to leaves are traversed. 3. We then form the union of the results of traversing all paths.

Given the dominator tree in Figure 10, the register allocation becomes:

$$
\begin{array}{rcl}
\multicolumn{3}{c}{From\ B0\ down\ to\ B5} \\
a : int & \leftarrow & R[1] \\
b : int & \leftarrow & R[2] \\
c : long & \leftarrow & R[3] \\
d : long & \leftarrow & R[4] \\
\ldots & & \\
\multicolumn{3}{c}{Frome\ B0\ down\ to\ B10} \\
a : int & \leftarrow & R[1] \\
b : int & \leftarrow & R[2] \\
e : int & \leftarrow & R[3]
\end{array}
$$

Forming the union of the results of traversing all paths, we get $a : int \leftarrow R[1]$, $b : int \leftarrow R[2]$, $c : long \leftarrow R[3]$, $d : long \leftarrow R[4]$, $e : int \leftarrow R[3]$. The full algorithm is described in Appendix A.0.3.

3.4.4 TypeInterpreter

Before invoking TypeInterpreter, we need to prepare the instruction ordering that is referred to as *type-check sequence* earlier. The scope of variable v in this sequence covers only the instructions following the definition instruction of v. The type-check sequence is prepared as follows:

1. Initialize an empty type-check sequence. 2. Invoke a tree traversal on the dominator tree. During the traversal, if a block is found for the first time, all instructions in the block is appended to the type-check sequence. If the block is visited again, then a popfa instruction is appended to the type-check sequence. Figure 5(b) shows a such sequence.

The TypeInterpreter scans the instructions in the type-check sequence in linear fashion and interprets instructions according to its transition rules. If TypeInterpreter successfully interprets all instructions, then the sequence is well typed and the instructions in program order are well typed too.

The following is the transition system of the TypeInterpreter. For simplicity, we omit the rules for rewriting the stack-based wire format into the register-based internal format that are also part of our transition functions. The TypeInterpreter uses a number of storage areas for data, code and book-keeping. Each storage area is represented as mapping of indices $i \in \mathbb{N}$ onto values of the appropriate type.

1. **ConstPool** $\mathbf{C}$: contains constant value and type pairs – $\langle v, \tau \rangle$, Symbolically, $\mathbf{C}[i \mapsto \langle v, \tau \rangle]$.

2. **RegisterFile** $\mathbf{R}$: contains types. Symbolically, $\mathbf{R}[i \mapsto \tau]$.

3. **Frame Stack** $\mathbf{fs}$: is used for book-keeping scopes. Each block instruction creates a stack $\mathbf{Frame}$. The $\mathbf{Frame}$ contains the current scope information and a pointer to the previous $\mathbf{Frame}$; it has the following four components: **MaxRegPointer** mrp is the maximum register number valid in the current block ($\mathbf{R}(1)$ to $\mathbf{R}(mrp)$ are visible in the current block). **Frame pointer** fp to the previous frame. An **origTypeStack** $\mathbf{ots}$ with its stack pointer $otsp$ is used to save and trace the original types of retyped registers in current block. Symbolically, $\mathbf{Frame} ::= (otsp)(\mathbf{ots})(mrp)(fp)$ and $\mathbf{fs}[i \mapsto \mathbf{Frame}]$.

4. **TypeOpStack** $\mathbf{tos}$: contains index and type pairs – $\langle j, \tau \rangle$. where j is the index to **RegisterFile** and $1 \leq j \leq mrp$, if $j = 0$, then the type is loaded from **ConstPool**. stack operation can be modelled by a combination of adding (or substracting) a constant to (from) its stack ointer $tosp$ and/or updating the mapping $\mathbf{tos}[i \mapsto \langle j, \tau \rangle]$.

Runtime Enviroment:$\mathbf{Env} ::= \langle \mathbf{fs}, \mathbf{tos}, tosp, \mathbf{R}, \mathbf{C}, mrp, \tau_{result} \rangle$. We do not list the program counter pc explicitly since it increases monotonically by one after each instruction.

3.4.5 Transition Rules

The rules for **decl** below reveal most aspects of the notation that we are using. The semantics of an instruction are defined by an axiom or an inference rule. A rule has a number of premises (above the horizontal line) and a conclusion. An axiom has a conclusion but no premises. Rules and axioms may have side conditions.

$$[\mathbf{decl}] \quad \frac{\mathbf{Env} \vdash \mathbf{R}[mrp+1 \mapsto \tau] \Rightarrow \mathbf{R}'}{\mathbf{Env} \vdash \langle \mathsf{decl}\ \tau, mrp, \mathbf{R} \rangle \Rightarrow \langle mrp+1, \mathbf{R}' \rangle}$$

The configuration on the left hand side of the $\Rightarrow$ consists of an instruction and its operands (e.g. decl), the current maximum register pointer (mrp), and the register file ($\mathbf{R}$). The configuration on the right hand side consists of the next value of the maximum register pointer (eg. $mrp + 1$) and the new register file ($\mathbf{R}'$). The notation $\mathbf{R}[mrp + 1 \mapsto \tau]$ extends the mapping $\mathbf{R}$ with a new domain/range pair. Any previous association for the new domain value $mrp+1$ is lost. It follows that it is sufficient to decrement the maximum register pointer to '*forget*' mappings for particular values in the domain.

The CAM instructions can be classified as:

1. **type initialization operations**: (decl, param) These two instructions initialize registers with types. decl τ assigns to the next available register in register file the type τ and increases mrp by one to point to the new assigned register. param has the same semantics as decl, but param can only be used in block $\mathbf{B0}$.

2. **type operand stack operations**: (loadr, loadc) The only two instructions that can increase the stack pointer $tosp$. loadr i: pushes a $\langle i, \tau \rangle$ onto the top of the stack, with type τ denoting the type of the ith register. loadc i: performs the equivalent operation for ith constant from the $\mathbf{ConstPool}$, but it pushs $\langle 0, \tau \rangle$, with τ denoting the type of the ith constant.

3. **stack frame operations**: (block, popfa) manage scopes during the type-level abstract interpretation. block i: pushes the current $\mathbf{Frame}$, which contains the current scopes, onto the stack, flushes $\mathbf{tos}$ and increases the frame pointer fp by one to point to a new $\mathbf{Frame}$. popfa: restores the previous $\mathbf{Frame}$ and sets the frame pointer fp to the next frame.

4. **apply operations**: (i2l, iadd, imul, move etc.): These operations are uniformly represented as a abstract data type $\mathsf{t}_{\mathsf{abstract}} = \tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}$. For example, i2l with abstract data type $int \rightarrow long$, iadd with abstract data type $int \times int \rightarrow int$, move with abstract data type $\tau \rightarrow \tau$, where τ is a type variable. For simplicity, we omit Env in the following rule.

$$\frac{\begin{array}{l}\vdash \mathsf{t}_{\mathsf{abstract}} : (\tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}) \\ \vdash (\mathbf{tos}(tosp), ..., \mathbf{tos}(tosp - n + 1)) \Rightarrow (\langle i_1, \tau_1' \rangle, ..., \langle i_n, \tau_n' \rangle) \\ \vdash \mathbf{tos}(tosp - n) \Rightarrow \langle i_{n+1}, \tau_{n+1'} \rangle \\ \vdash (\tau_1' \trianglelefteq \tau_1 \wedge ... \wedge \tau_n' \trianglelefteq \tau_n) \\ \vdash (0 < i_{n+1} \wedge \tau_{n+1} \trianglelefteq \tau_{n+1}')\end{array}}{\vdash \langle \mathsf{apply}\ \mathsf{t}_{\mathsf{abstract}}, tosp \rangle \Rightarrow \langle tosp - n + 1 \rangle}$$

[**apply**]

$$\frac{\begin{array}{l}\vdash \mathsf{t}_{\mathsf{abstract}} : (\tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}) \\ \vdash (\mathbf{tos}(tosp), ..., \mathbf{tos}(tosp - n + 1)) \Rightarrow (\langle i_1, \tau_1' \rangle, ..., \langle i_n, \tau_n' \rangle) \\ \vdash \mathbf{tos}(tosp - n) \Rightarrow \langle i_{n+1}, \tau_{n+1'} \rangle \\ \vdash (\tau_1' \trianglelefteq \tau_1 \wedge ... \wedge \tau_n' \trianglelefteq \tau_n) \\ \vdash (0 < i_{n+1} \wedge \tau_{n+1} \not\trianglelefteq \tau_{n+1}') \\ \vdash \mathbf{ots}[otsp + 1 \mapsto \langle i_{n+1}, \tau_{n+1}' \rangle] \Rightarrow \mathbf{ots}' \\ \vdash \mathbf{R}[w \mapsto \tau_{n+1}] \Rightarrow \mathbf{R}'\end{array}}{\vdash \langle \mathsf{apply}\ \mathsf{t}_{\mathsf{abstract}}, tosp, \mathbf{ots}, otsp, \mathbf{R} \rangle \Rightarrow \langle tosp - n + 1, \mathbf{ots}', otsp + 1, \mathbf{R}' \rangle}$$

   If $0 < i_{n+1}$, then i_{n+1} is loaded from the $\mathbf{R}$, hence it's updateable. $\tau_1 \trianglelefteq \tau_2$ denotes that τ_1 is a subtype of τ_2. Each fun has two transition rules, the one above is the transition rule for $\tau_i' \trianglelefteq \tau_i$, where $i = 1, ..., n$ and $\tau_{n+1} \trianglelefteq \tau_{n+1}'$. If $\tau_{n+1} \not\trianglelefteq \tau_{n+1}'$, then the $\langle i_{n+1}, \tau_{n+1}' \rangle$ need to be pushed onto the $\mathbf{tos}$ in the current $\mathbf{Frame}$ and will be restored when current $\mathbf{Frame}$'s (scope) is popped.

5. **array length operation**: (arraylength) pops out two pairs $\langle i_1, \tau_1 \rangle$ and $\langle i_2, \tau_2 \rangle$ from $\mathbf{tos}$. If $\tau_1 \notin \tau_{array}$ or $\tau_2 \neq int$, then we have a type mismatch.

6. **miscellaneous**: (result, retnv) result τ: initilizes τ_{result} with τ. retnv: looks at the $\langle i, \tau \rangle$ pair on $\mathbf{tos}$. If $\tau \neq \tau_{result}$ then we have a type mismatch error.

The formal semantics of above operations are listed in Appendix B.

3.5 Control-Flow Safety Analysis

Branches that cause the flow of control to leave a basic block must have *valid* targets. Intuitively [12, 25], this rules out jumping into the middle of an instruc-

tion or to data as if it were code. More subtly, we need to also rule out jumping to an instruction that causes the data flow analysis of the Bytecode Verifier to fail. Knowing all variables and their types, the property $\mathbf{VALID}(b)$ at the entry of block b is sufficient to check the well-typedness of instructions in block b. If a $\mathbf{VALID}(b)$ ensures the well-typedness of instructions in block b, then this $\mathbf{VALID}(b)$ is called well-typed. Annotating each basic block with a well-typed $\mathbf{VALID}$ at compiler time is the technique used by the KVM [22] to reduce the verification cost at runtime. A control flow safety definition based on well-typedness and VALID is given below:

Definition 2 (Valid control flows to block b) *If the* $\mathbf{VALID}(b)$ *calculated based on the control flows to block b fails to ensure the well-typedness of instructions in block b, then those control flows to block b are invalid with respect to these instructions in block b, otherwise the control flows to the block are valid.*

Having the control flow safety of individual blocks, it is no longer difficult to define the control flow safety of a program.

In section 3.4, $\mathbf{VALID}(b)$ is recursively defined on dominance. If the control flows to block b preserves the dominance on which $\mathbf{VALID}(b)$ is determined,then it also preserves $\mathbf{VALID}(b)$.

3.5.1 Dominance Invariance

We have successfully tied the valid variable analysis and control flow safety analysis together by dominance. Dominance plays a fundamental role in the certification of CAM. Its control flow safety check relies on whether a control flow graph comforms to the dominance on which virtual register allocation (section 3.4.3) is based.

The dominance of a control flow graph is summarized as follows: Let $dmtree : \mathbb{G} \rightarrow \mathbb{T}$ denote the algorithm to compute the dominator tree from the CFG. $dmtree$ is a function which maps a control flow graph to an unique dominator tree, while $dmtree^{-1}$ is not a function. Control flow safety verification requires to check whether a control flow graph G conforms to a given dominator tree T. An intuitive way is to calculate the dominator tree T' of G and to compare T' and T. $dmtree$ functions are usually not linear. Alstrup *et al.* [2] published a theoretical linear-time complexity dominance algorithm, but the actual complexity of the algorithm, using practical data structures, is $O(E + Vloglogloglog V)$. Cooper's work [4] on fast dominance algorithms has demonstrated that a well-engineered

$O(V^2)$ dominance algorithm runs faster, in practice, than the classic Lengauer-Tarjan algorithm [11], which has a timebound of $O(E * log(V))$. This means that improvements [2, 8, 11] in asymptotic complexity may not help on realistically-sized examples. Stephen [1] has given a practical $O(E + V)$ dominance algorithm limited to reducible graphs.

In previous work [24], we gave a simple $O(V + E)$ algorithm, from a program certifi cation viewpoint, to verify whether a control flow graph conforms to a given dominator tree. The algorithm demonstrates that checking the correctness of solution is easier or less complicated than fi nding the solution in the fi rst place. The central idea is briefly repeated here.

An edge is *dominance invariant* if after adding the edge to a control flow graph, the resulting control flow graph has the same dominator tree as the original control flow graph.

Definition 3 (dominance invariant successors) *The **dominance invariant successors** of node z, denoted by $\mathbf{domis}(z)$, is the set of nodes y and the edge from z to y is dominance invariant.*

$$\mathbf{domis}(z) \stackrel{def}{=} \{y | dmtree(G'(V, E_G \cup \{(z, y)\})) \equiv dmtree(G(V, E_G))\}$$

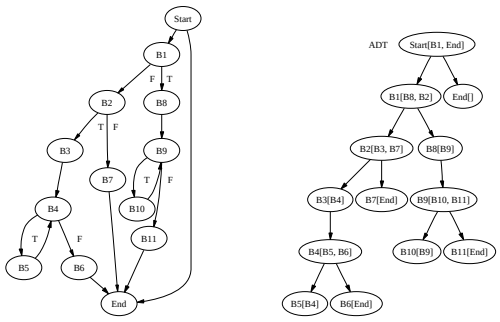
Lemma 1 *$\forall z \in V - \{start\}$, $\mathbf{domis}(z)$ is the set of nodes whose immediate dominator (parent) dominate z.*

By the defi nition of the dominator tree, the immediate dominator of node x is also the parent of node x. An edge (z, x) is dominance invariant if and only if the parent of x is the ancestor of z. The proof is given in Appendix A.0.7.

3.5.2 Augmented Dominator Tree

Given a control flow graph G and its dominator tree T, we annotate each tree node with its successors. The resulting dominator tree is called *Augmented Dominator Tree*. Where G_{adj} is the adjacency list encoding of the control flow graph G, $ADT(T, G_{adj})$ is the control flow safety certifi cate transported in CAM. An example encoding is shown in Figure 11. Verifi cation proceeds as follows:

Firstly, we verify that each successor list contains valid nodes. For any tree node x except $start$, let $successor(x)$ denote the successor list of x in the ADT. By Defi nition 3, if $successor(x) \subseteq domis(x)$, then $successor(x)$ is valid. By



(a) (b)

Figure 11: (b) is the ADT encoding of the control flow graph (a)

Lemma 1, if the parent of node in $successor(x)$ dominates x, then $successor(x)$ is valid. An O(V + E) ADTSuccessorVerifier algorithm is given in Appendix A.0.4.

Secondly, we verify that G_{adj} is a valid CFG (refer to the definition of a control flow graph in section 2.1). This is not obvious since $dmtree^{-1}$ is not a function. Figure 12 shows multiple CFGs corresponding to one dominator tree. In other words, G_{adj} is not unique. We define an *abstract control flow graph* denoting $\mathbb{G}_{abs}(T)$. It is the abstraction of all control flow graphs conformming to the dominator tree T. $\mathbb{G}_{abs}(T)$ is constructed from the dominator tree T by linking all leaf nodes to end node. Every edge (x, y) in $\mathbb{G}_{abs}(T)$ models a flow path from x to y and x isn't neccessary directly connected to y. All control flows conforming to the dominator tree T are modeled in $\mathbb{G}_{abs}(T)$ because if x dominates y, then x appears on every path from $start$ to y and end is reachable from any node. It's easy to verify that $\mathbb{G}_{abs}(T)$ in Figure 12(g) abstractly models the control flow in CFGs from Figure 12(b) to 12(f).

A CFG G is said to conform to $\mathbb{G}_{abs}$, if for all edges $(x, y) \in \mathbb{G}_{abs}$, there is a path from x to y in G. With the abstract control flow graph, we can give a definition of the validity of G_{adj}.

Definition 4 (Validity of G_{adj}) *Given an $ADT(T, G_{adj})$, G_{adj} is a valid CFG if G_{adj} conforms to $\mathbb{G}_{abs}(T)$.*

An elimination strategy is used to verify that G_{adj} conforms to $\mathbb{G}_{abs}(T)$. For all edges $(x, y) \in \mathbb{G}_{abs}(T)$, if y appears on any path from x to end in the CFG represented by G_{adj}, then we delete the edge (x, y) from $\mathbb{G}_{abs}(T)$. If no edge is left in $\mathbb{G}_{abs}(T)$ then G_{adj} is a valid adjacent list encoding of the control flow graph. This procedure has O(V + E) complexity.

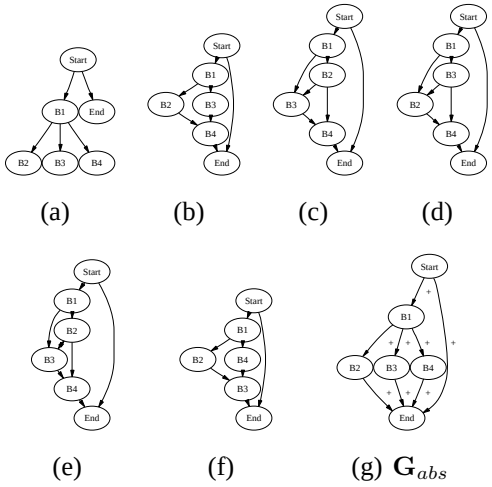


Figure 12: Control flow graphs (b), (c),(d), (e) and (f) have the same dominator tree (a). (g) The abstract model of all control flow graphs, edge $\xrightarrow{+}$ represents a control flow path

Theorem 1 (Validity of ADT) *An $ADT(T, G_{adj})$ is valid if and only if*

1. $\forall x \in V - \{start\}, \forall z \in successor(x), z \in domis(x)$.
2. G_{adj} *conforms to* $\mathbb{G}_{abs}(T)$.

The ADTVerifier consists of the ADTSuccessorVerifier and the verifier of G_{adj}. Both have O(V + E) complexity so the ADTVerifier is linear. The ADTVerifier guarantees only that $ADT(T, G_{adj})$ is valid with respect to the dominator tree T. Finally, we need to make sure that the control flows in G_{adj} match with the $\mathbf{CtrlInst}$ at the end of each basic block (node). This check is linear over the size of the basic block.

4 Implementation Overview

JVML to CAM compilation: We implement a compiler that converts existing JVML code into CAM code. The .cam file keeps the original class file format and only replaces JVML's code attribute by the code attribute of CAM instructions. An additional attribute ADT is added that encodes the augmented dominator tree.

The compilation procedure first recovers all nested *try-catch* blocks. All blocks in a *try* or *catch* block are grouped as a sub control flow graph. Each *try* or *catch* block is treated as a single block in the global control flow graph of a method. Dominator trees of the global control flow and sub control flow graphs nested in a *try* or *catch* are calcuated seperately. The complete dominator tree is fromed by replacing *try* or *catch* nodes with the dominator tree of the control flow graph nested within it.

After this step, fixpoint iterative data analysis is invoked to recover all variables and their types. The virtual register allocation algorithm assigns registers to all variables and *decl* instructions are inserted at the appropriate blocks. Finally, all JVML instructions are rewritten into CAM instructions.

CAMVerifer: The CAMVerfier consist of logically independent ADTVerifier and TypeInterpreter. The former in turn consists of an ADTSuccessorVerifier and a verifier for G_{adj}. Preparing the type-check sequence is not strictly necessary, since our TypeInterpreter can traverse the dominator tree directly. Since the ADTSuccessorVerifier is a tree traversal based algorithm, our TypeInterpreter and ADTSuccessorVerifier are implemented as a single integrated algorithm. The verifier of G_{adj} needs to know the complete control flow graph, so it is implemented as an independent component.

5 Related work

Checking the well-typedness of a program is a well-studied problem at different levels: high-level source code, abstract machine code, native machine code. Existing approaches include syntax-directed checking, abstract interpretation, dataflow analysis, general logic, and type theory.

On the high level source code side, most modern programming languages have structured grammars. Syntax-directed type checking is sufficient to check the well-typedness of source code in these languages. Type-checkers have been widely implemented in compilers for strongly typed languages.

Abstract machines [6] simulate real hardware machines by allowing step-by-step execution. They bridge the semantic gap between high level languages and low level real machines. The Java Virtual Machine is the abstract machine for Java. The Java compiler produces stack-based JVML code that can be verified by the Java Bytecode Verifier before execution with some basic level safety (including well-typedness).

On the other extreme, verifying the semantics of low-level native machine

code conforming to the semantics of high level source code is not trivial and less efficient. However, encouraging progress has been made in recent years:

- *Proof Carrying Code* [14] uses first-order logic to represent correctness properties and a safety proof is constructed based on a general system such as Edinburgh LF and carried along with code.
- *Typed Assembly Language* [13] preserves typing information from a high-level program written in a strongly-typed language and includes it with the compiled program. It can then be checked by an ordinary type checker.

The JVM is still the most widely used approach for mobile code. Unfortunately, the design of JVML makes bytecode verification relatively complex in both time and space consumption. KVM [22] for resource limited devices adopts a lightweight verification algorithm [9] instead of the full Bytecode Verifier.

Kozen's [10] work on Efficient Code Certification (ECC) has a nature context-free structure which mirrors the structure of high-level functional languages. EC-Cissimilar to SafeTSA (introduced in the Introduction) in the sense that bothdepend on the high-level control structure. The structure of ECC consists of well-nested intervals of instructions, called blocks. Its certificate consists a block tree. The block tree mirrors the nested structures of functional languages. ECC limits its inputs tofunctional source code. Therefore, it does not need to deal with arbitrary jumps (even irreducible control flow graphs) as our approach does.

Davis [5] experiments with converting the stack-based JVML into a register basedinstruction set. His experiment shows that the transformation reduces the number of executed instructions by 34.88% and increases the number of operand fetch instructions by 44.81%.One of the main costs of an interpreter is that for instruction dispatch. Davis' work has demonstrated that virtual register machines are an attractive alternative to virtual stack machines.

Rose and Rose [18] propsed a (sparse) annotation of JVM code with types to enable a one-pass verification of well-typedness. Roughly speaking, this transforms a type reconstruction problem into a type checking problem, which is easier. More precisely, the type inference problem is a data flow analysis problem that requires an iterative solution, whereas the type checking problem merely needs a single pass to check consistency of the type annotations with the code. It trades space for time: it is sufficient to store only the state type for the entry point to each basic block because the remaining state types in that block can be computed in linear time.

Sreedhar [20, 21, 19] proved a weaker version of lemma 1. D-J Graphs are an alternative program representation used in loop identification, elimination-based data flow analysis and for placing ϕ nodes in linear time.

6 Conclusion

We have designed an alternative imperative core for the Java Virtual Machine that aims to encode well-typedness as compactly and transparently as possible. We apply the data flow analysis to the design of CAM. The introducing of dominance as code certificate greatly simplify the verification logic and successfully eliminate the requirement of data flow analysis for well-typedness check.

7 Acknowledgement

This research effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research (ONR) under agreement N00014-01-1-0854. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, ONR, or any other agency of the U.S. Government.

A Algorithms and Proofs

A.0.3 Register Allocation

proc $\mathbf{RegisterAllocater}$
initialize all node as white
$stack \leftarrow \{\}$
$x \leftarrow B0$
$MaxRegNo \leftarrow 0$
for $(v \in DEF(x))$ **do**
$MaxRegNo \leftarrow MaxRegNo + 1$

```
      assign R[MaxRegNo] to v
  od
  push(stack, ⟨x, MaxRegNo⟩)
  while (true) do
        for (y ∈ children(x)) do
            if (y is white)
              then
                    for (v ∈ DEF(y)) do
                        MaxRegNo ← MaxRegNo + 1
                        assign R[MaxRegNo] to v
                    od
                    mark y as black
                    push(stack, ⟨y, MaxRegNo⟩
                    x ← y
                    break
            fi
        od
        pop(stack)
        if (stack ≠ {})
          then ⟨x, MaxRegNo⟩ ← peek(stack)
           else return
        fi
  od
```

A.0.4 ADTSuccessorVerifier

```
proc ADTSuccessorVerifier
  initialize all nodes as white
  stack ← {}
  x ← B0
  push(stack, x)
  while (true) do
        for (y ∈ children(x)) do
            if (y is white)
              then
                    for (z ∈ unchkedPred(y)) do
                        if (parent(z) ∉ stack)
                          then return false fi
                    od
```

```
                mark y as black
                push(stack, y)
                x ← y
                break
            fi
        od
        if (successor(x) ≠ Ø)
          then for (z ∈ successor(x)) do
                if (z is white)
                  then
                      unchkedPred(z) ← unchkedPred(z) ∪ {x}
                elsif (parent(z) ∉ stack)
                    then return false
                fi
              od
        fi
        pop(stack)
        if (stack ≠ {})
          then x ← peek(stack)
           else return true
        fi
  od
```

A.0.5 InsertMonitor

```
proc InsertMonitor
  initialize all nodes as white
  stack ← {}
  x ← B0
  push(stack, x)
  while (true) do
        for (y ∈ children(x)) do
            if (y is white)
              then
                  for (z ∈ patchexit(y)) do
                      if (parent(z) ≠ parent(y)) then insert monitorexit into y
                        od
                        mark y as black
                        push(stack, y)
```

```
                    x ← y
                    if (y is synch)
                      then insert monitorenter into y
                    fi
                    break
                fi
            od
            if (successor(x) ≠ ∅)
              then for (z ∈ successor(x)) do
                    if (z is white) then patchexit(z) ← patchexit(z) ∪ {x}
                    elsif (parent(z) ≠ x) then insert monitorexit into z
                    fi
                od
            fi
            pop(stack)
            if (stack ≠ {})
              then x ← peek(stack)
               else return
            fi
        od
```

A.0.6 CheckJsr

```
proc CheckJsr
  initialize all nodes as white
  stack ← {}
  x ← B0
  push(stack, x)
  while (true) do
      for (y ∈ children(x)) do
        if (y is white)
          then
              for (z ∈ patchexit(y)) do
                if (parent(z) ≠ parent(y)) then insert monitorexit into y
                od
                mark y as black
                push(stack, y)
                x ← y
                if (y has jsr)
```

```
                then insert jsr into y
              fi
              break
        fi
    od
    if (successor(x) ≠ ∅)
      then
          if (|successor(y)| = 2)
            v ← remove successor(y);
            if (parent(v) ≠ parent(y)) then return false
            fi
      fi
      if (|successor(y)| = 2 ∧ successor(y).2nd = successor(x))
        then
            if (z is white) then patchexit(z) ← patchexit(z) ∪ {x}
            elsif (parent(z) ≠ x) then insert monitorexit into z
            fi
            then
      fi
      pop(stack)
      if (stack ≠ {})
        then x ← peek(stack)
        else return
      fi
    od
```

A.0.7 Proof of lemma 1

PROOF. First we show that if $idom(y)$ strictly dominates z, then $y \in domis(x)$. Let $x = idom(y)$, the CFG edge (z, y) might affect the dominance relation between x and z, y and the dominator subtrees rooted at z, y respectively. We divide the proof in several cases, see the dominator tree in Figure 13

1. **Subtree rooted at z will not be changed:** The new CFG edge (z, y) bypasses all $descendant(z)$, which are strictly dominated by z.

2. **Subtree rooted at y will not be changed:** The new CFG edge (z, y) introduces only new incoming paths to y, and y still appears on every path from $start$ to $\forall v \in descendant(y)$.

3. **Subtree rooted at x will not be changed:** Both z and y are strictly dominated by x. As x appears on every path from $start$ to z, so x will also appear on path

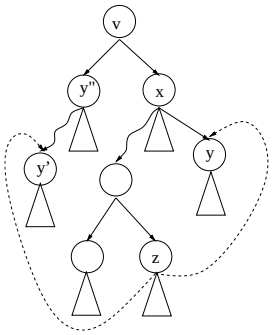


Figure 13: Solid edges are tree edges, curve edges are tree paths and dash edges are new edges added to the original control flow graph. Triangles represent subtree. Given the dominator tree. adding edge (z, y) to its control flow graph G will not render a different dominator tree in contrast to adding (z, y') which render a new dominator tree with node y' as the child of root.

$start \rightarrow x...z \rightarrow y$. Hence, x still immediate dominates y.

We conclude that the subtree rooted at x is same in all cases and that the dominance relation is preserved.

Secondly, we need to show that no other nodes can be found in $domis(z)$. Suppose $y' \in domis(z)$, $idom(y') \not\prec z$, and let $y'' = idom(y')$. Then we can find a path $start \rightarrow ...z \rightarrow y'$(Figure 13) that bypasses y'', so y'' no longer appears on every path from $start$ to y'. As y'' does not dominate y', this contradicts the assumption that $y' \in domis(z)$.

B Operational semantics

For simplicity, we omit $\mathbf{Env}$ in the following confi guration.

type initialization operations

$$[\textbf{decl}]\quad \frac{\mathbf{Env} \vdash \mathbf{R}[mrp+1 \mapsto \tau] \Rightarrow \mathbf{R}'}{\mathbf{Env} \vdash \langle \mathsf{decl}\ \tau, mrp, \mathbf{R}\rangle \Rightarrow \langle mrp+1, \mathbf{R}'\rangle}$$

$$[\textbf{param}]\quad \frac{\vdash \mathbf{R}[mrp+1 \mapsto \tau] \Rightarrow \mathbf{R}'}{\vdash \langle \mathsf{param}\ \tau, mrp, \mathbf{R}\rangle \Rightarrow \langle mrp+1, \mathbf{R}'\rangle}$$

frame stack operations

$$[\textbf{block}]\quad \frac{\vdash \mathbf{fs}[fp+1 \mapsto \mathbf{Frame}(otsp)(\mathbf{ots})(mrp)(fp)] \Rightarrow \mathbf{fs}'}{\vdash \langle \mathsf{block}\ i, \mathbf{fs}, fp, tosp\rangle \Rightarrow \langle \mathbf{fs}', fp+1, tosp - tosp\rangle}$$

$$[\textbf{popfa}]\quad \frac{\begin{array}{l}\vdash \mathbf{R} \oplus \{\mathbf{ots}(otsp - i + 1) \Rightarrow \langle n_i, \tau_i\rangle, n_i \mapsto \tau_i | i \leftarrow [otsp...1]\} \\ \vdash \mathbf{fs}(fp) \Rightarrow \mathbf{Frame}(otsp')(\mathbf{ots}')(mrp')(fp')\end{array}}{\vdash \langle \mathsf{popfa}, \mathbf{fa}, fp, otsp, \mathbf{ots}, mrp, \mathbf{R}\rangle \Rightarrow \langle \mathbf{fa}', fp', otsp', \mathbf{ots}', mrp', \mathbf{R}'\rangle}$$

type operand stack operations

$$[\textbf{loadr}]\quad \frac{\begin{array}{l}\vdash 0 < i \leq mrp \\ \vdash \mathbf{R}(i) \Rightarrow \tau \\ \vdash \mathbf{tos}[tosp+1 \mapsto \langle i, \tau\rangle] \Rightarrow \mathbf{tos}'\end{array}}{\vdash \langle \mathsf{loadr}\ i, tosp, \mathbf{tos}\rangle \Rightarrow \langle tosp+1, \mathbf{tos}'\rangle}$$

$$[\textbf{loadc}]\quad \frac{\begin{array}{l}\vdash 0 \leq i \leq \mathbf{C}_{max} \\ \vdash \mathbf{C}(i) \Rightarrow \tau \\ \vdash \mathbf{tos}[tosp+1 \mapsto \langle 0, \tau\rangle] \Rightarrow \mathbf{tos}'\end{array}}{\vdash \langle \mathsf{loadc}\ i, tosp, \mathbf{tos}\rangle \Rightarrow \langle tosp+1, \mathbf{tos}'\rangle}$$

switch operation

$$[\textbf{switch}]\quad \frac{\begin{array}{l}\vdash \mathbf{tos}(tosp) \Rightarrow \langle i, \tau_1\rangle \\ \vdash \tau_1 = int\end{array}}{\vdash \langle \mathsf{switch}, tosp\rangle \Rightarrow \langle tosp - 1\rangle}$$

array length operation

$$[\textbf{arraylength}]\quad \frac{\begin{array}{l}\vdash (\mathbf{tos}(tosp), \mathbf{tos}(tosp-1)) \Rightarrow (\langle i, \tau_1\rangle, \langle j, \tau_2\rangle) \\ \vdash \tau_1 \in \tau_{array} \wedge \tau_2 = int\end{array}}{\vdash \langle \mathsf{arraylength}, tosp\rangle \Rightarrow \langle tosp - 1\rangle}$$

apply operation

$$[\textbf{apply}]\quad \frac{\begin{array}{l}\vdash \mathsf{t}_{\mathsf{abstract}} : (\tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}) \\ \vdash (\mathbf{tos}(tosp), ..., \mathbf{tos}(tosp - n + 1)) \Rightarrow (\langle i_1, \tau_1'\rangle, ..., \langle i_n, \tau_n'\rangle) \\ \vdash \mathbf{tos}(tosp - n) \Rightarrow \langle i_{n+1}, \tau_{n+1'}\rangle \\ \vdash (\tau_1' \trianglelefteq \tau_1 \wedge ... \wedge \tau_n' \trianglelefteq \tau_n) \\ \vdash (0 < i_{n+1} \wedge \tau_{n+1} \trianglelefteq \tau_{n+1}')\end{array}}{\vdash \langle \mathsf{apply}\ \mathsf{t}_{\mathsf{abstract}}, tosp\rangle \Rightarrow \langle tosp - n + 1\rangle}$$

$$[\textbf{apply}]\quad \frac{\begin{array}{l}\vdash \mathsf{t}_{\mathsf{abstract}} : (\tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}) \\ \vdash (\mathbf{tos}(tosp), ..., \mathbf{tos}(tosp - n + 1)) \Rightarrow (\langle i_1, \tau_1'\rangle, ..., \langle i_n, \tau_n'\rangle) \\ \vdash \mathbf{tos}(tosp - n) \Rightarrow \langle i_{n+1}, \tau_{n+1'}\rangle \\ \vdash (\tau_1' \trianglelefteq \tau_1 \wedge ... \wedge \tau_n' \trianglelefteq \tau_n) \\ \vdash (0 < i_{n+1} \wedge \tau_{n+1} \not\trianglelefteq \tau_{n+1}') \\ \vdash \mathbf{ots}[otsp+1 \mapsto \langle i_{n+1}, \tau_{n+1}'\rangle] \Rightarrow \mathbf{ots}' \\ \vdash \mathbf{R}[w \mapsto \tau_{n+1}] \Rightarrow \mathbf{R}'\end{array}}{\vdash \langle \mathsf{apply}\ \mathsf{t}_{\mathsf{abstract}}, tosp, \mathbf{ots}, otsp, \mathbf{R}\rangle \Rightarrow \langle tosp - n + 1, \mathbf{ots}', otsp + 1, \mathbf{R}'\rangle}$$

miscellaneous

$$[\textbf{result}]\quad \frac{\vdash \tau_{result} = \top}{\vdash \langle \mathsf{result}\ \tau\rangle \Rightarrow \langle \tau_{result} = \tau\rangle}$$

$$[\textbf{retnv}]\quad \frac{\begin{array}{l}\vdash \mathbf{tos}(tosp) \Rightarrow \langle i, \tau\rangle \\ \vdash i \leq mrp \\ \vdash \tau = \tau_{result}\end{array}}{\vdash \langle \mathsf{retnv}, \mathsf{tosp}\rangle \Rightarrow \langle tosp - 1\rangle}$$

The notation $\mathbf{R} \oplus \{\mathbf{ots}(otsp - i + 1) \Rightarrow \langle n_i, \tau_i\rangle, n_i \mapsto \tau_i | i \leftarrow [otsp...1]\}$

extends the mapping $\mathbf{R}$ with a set of new domain/range pair $n_i \mapsto \tau_i$, where i is from $otsp$ to 1 and $\langle n_i, \tau_i \rangle$ is popped out from $\mathbf{ots}$.

References

[1] S. Alstrup and P. W. Lauridsen. A simple and optimal algorithm for finding immediate dominators in reducible graphs. 1996.

[2] S. Alstrup, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *DIKU technical report*, (35), 1996.

[3] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on Static Single Assignment form. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 137–147. ACM Press, 2001.

[4] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. In *Software-Practice And Experience*, pages 4:1–10. John Wiley and Sons, Ltd., 2001.

[5] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The Case for Virtual Register Machines. In *Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49, San Diego, California, 2002. ACM Press.

[6] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[7] A. Gal, C. W. Probst, and M. Franz. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.

[8] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185–194, 1985.

[9] G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.

[10] D. Kozen. Efficient code certification. Technical Report TR98-1661, 1998.

[11] T. Lengauer and R.E.Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Trans. Program. Lang. Syst.*, volume 1, pages 115–120, July 1979.

[12] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, 1999.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language (extended version). Technical Report TR97-1651, 21, 1997.

[14] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[15] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[16] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, 1998.

[17] Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[18] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[19] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, 1995.

[20] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, Nov. 1996.

[21] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, Mar. 1998.

[22] Sun Microsystem. CLDC Specification, v1.1.

[23] J. von Ronne, N. Wang, A. Apel, and M. Franz. Virtual Machine for Interpreting Programs in Static Single Assignment Form. Technical Report 03-19, University of California, Irvine, School of Information and Computer Science, 2003.

[24] N. Wang, M. Franz, and N. Dalton. Enabling Efficient Program Analysis for Dynamic Optimization of a Family of Safe Mobile Code Formats. Technical Report 02-24, University of California, Irvine, School of Information and Computer Science, 2002.

[25] F. Yellin. Low Level Security in Java, 1995.