A Portable Virtual Machine Target For Proof-Carrying Code

Michael Franz [a,*], Deepak Chandra [a], Andreas Gal [a], Vivek Haldar [a], Christian W. Probst [a], Fermín Reig [b], Ning Wang [a]

[a] *Donald Bren School of Information and Computer Science, University of California, Irvine*

[b] *School of Computer Science, University of Nottingham*

Abstract

Virtual Machines and Proof-Carrying Code are two techniques that have been used independently to provide safety for (mobile) code. Both these techniques have strengths and individual limitations. Existing virtual machines, such as the Java VM, have several drawbacks: First, the effort required for safety verification is considerable. Second, and more subtly, the need to provide such verification by the code consumer inhibits the amount of optimization that can be performed by the code producer. This in turn makes just-in-time compilation surprisingly expensive. Proof-Carrying Code, on the other hand, has its own set of limitations, among which are the size of proofs and the fact that the certified code is no longer machine-independent. By combining the two techniques, we are able to overcome these limitations. Our hybrid safe-code solution uses a virtual machine that has been designed specifically to support proof-carrying code, while simultaneously providing efficient just-in-time compilation and target-machine independence. In particular, our approach reduces the complexity of the required proofs, resulting in fewer proof obligations that need to be discharged at the target machine.

Key words: Virtual Machines, Proof-Carrying Code, Safe Mobile Code, Dynamic (Just-In-Time) Compilation
PACS:

---

* Corresponding author.
Email address: franz@uci.edu (Michael Franz).

1 Introduction

A considerable amount of effort has recently been invested into (mobile) code safety. The general idea is simple: rather than *trusting* a piece of code because it came from a specific provider (for example, because it was purchased in a box in a reputable store or because it was digitally signed), the code consumer *verifies* the code prior to execution. Verification means determining the code's safety *by examining the code itself* rather than *where it came from.*

Over the past few years, three main approaches to safe code [1] have been developed. They are, in turn, *virtual machines with code verification* [1], *Proof-Carrying Code* [2], and *inherently safe code formats* [3–5].

In virtual machines (VMs) with code verification, the code is examined to ensure that the semantic gap between the source language and the virtual machine instruction format is not exploited for malicious purposes. For example, virtual machines have general `goto` instructions that in principle allow to jump to an offset that is the middle of an instruction. Some control flows, including the aforementioned, are deemed illegal by the Java Virtual Machine (JVM) specification [2]. Therefore, in the JVM a verifier needs to make sure that the target of a jump is the beginning of an instruction. As another example, the language definition of Java requires every variable to be initialized before its first use—unless control flow is strictly linear, this property cannot be inferred trivially from the virtual machine program but requires the verifier to perform data flow analysis.

In proof-carrying code solutions, the code producer attaches a *safety proof* to an executable. Upon receiving the code, the recipient examines it and calculates a *verification condition* from the code. The verification condition relates to all the potentially unsafe constructs that actually occur in the executable. It is the task of the code producer to supply a proof that discharges this verification condition, or else the code will not be executed.

In the third approach, an *inherently safe code format* is used to transport the mobile program, making most aspects of program safety a well-formedness criterion of the mobile code itself. Checking the well-formedness of such a

[1] We much prefer the term *safe code* to the term *mobile code,* for two reasons. First, all code is becoming "mobile", with program patches and whole applications increasingly being distributed via the Internet. Second, we believe that in the not so far future, *all* code resident on desktop computers (outside of a small hardware-secured trusted computing base) will be verified prior to every execution.

[2] We note that code obfuscators often work by producing irreducible control flows that, while verifiable, do not easily map back onto the control structures of the source language.

format is much simpler than verifying bytecode. The disadvantage is that a much more complex and memory-intensive machinery is required at the code recipient's site, as inherently safe formats are based on compression using syntax and static program semantics. As a consequence, this approach is less well suited for resource-constrained client environments.

In the following, we describe our hybrid approach that combines the first two solutions and applies elements of the third. The aim of our research is to find the "sweet spot" reconciling high execution performance of the final code, high dynamic compilation efficiency, small proof size of the proof-carrying code component, and limited resource consumption on the client computer.

The article is structured as follows: First, we discuss the Java Virtual Machine, not only because it is a good example of a VM, but also because it is the de-facto standard for transporting mobile code. Then, we give an overview of proof-carrying code (Section 3). Section 4 presents the case for a new virtual machine specifically designed for proof-carrying code. Section 5 gives a sketch of the architecture we are currently implementing. This section also contains several examples. Section 6 expands on the proof-carrying code aspects of our work and continues the examples from the previous section. After presenting related work in Section 7, we give an outlook to future work and conclude the article.

2 The Java Virtual Machine

The Java Virtual Machine's bytecode format ("Java bytecode") has become the de-facto standard for transporting mobile code across the Internet. However, it is generally acknowledged that Java bytecode is far from being an ideal mobile code representation—a considerable amount of preprocessing is required to convert Java bytecode into a representation more amenable to an optimizing compiler, and in a dynamic compilation context this preprocessing takes place while the user is waiting. However, there are additional limitations, some of which we present in this section.

First, it has been shown that the rules for bytecode verification do not exactly match those of the Java Language Specification, so that there are certain classes of perfectly legal Java programs that are rejected by all compliant bytecode verifiers [6]. This happens because the Java bytecode verification algorithm used is expected to compute a unique abstraction for each instruction but is unable to do so for complex control flows that are, e.g., side-effects of the `jsr` instruction. Figure 1 shows the code for an example (from [6]) of a legal Java program that is compiled to bytecode that is rejected by the verifier. Here, the verifier is not able to determine that the variable `i` is initialized

```
class Test1 {
      int test(boolean b) {
          int i;
          try {
              if (b) return 1;
              i=2;
          } finally { if (b) i=3; }
          return i;
      }
}
```

Fig. 1. A legal Java program that is compiled to bytecode that is rejected by the verifier. The verifier is not able to determine that the variable `i` is initialized along all paths that reach `return i`, and thus rejects it. This is caused by the fact that `return 1` executes the `finally` block before returning. Thus, the verifier enters the `finally` block along a path where `i` is not defined. Then, the rest of the code is verified and the `finally` block is reached along the path that defines `i=2`. Since the assignment `i=3` is only executed conditionally, the verifier determines that `return i` might be reached with an undefined variable `i`.

along all paths, and thus rejects it.

Second, due to the need to verify the code's safety upon arrival at the target machine, and also due to the specific semantics of the JVM's particular security scheme, many possible optimizations cannot be performed in the source-to-Java bytecode compiler, but can only be done at the eventual target machine—or at least they would be very cumbersome to perform at the code producer's site.

For example, information about the redundancy of a type check may often be present in the front-end (because the compiler can prove that the value in question is of the correct type on every path leading to the check), but this fact cannot be communicated safely in the Java bytecode stream and hence needs to be re-discovered in the just-in-time compiler. By *communicated safely* we mean in such a way that a malicious third party cannot construct a mobile program that falsely claims that such a check is redundant.

Another example is common subexpression elimination, which due to verification can only be performed at the code consumer. A compiler generating Java bytecode could in principle perform common subexpression elimination and store the resulting expressions in additional, compiler-created local variables. However, this approach is incomplete because it cannot factor out address calculations (for arrays etc.)—since verification requires preserving the language abstractions, such optimizations for the JVM can be performed only *after* the code has been verified (i.e., on the target machine).

Another problem is that just-in-time compilation occurs while interactive users may be waiting for execution to commence. If dynamic compilation time were unbounded, one would be able to perform extensive optimization and obtain a high code quality. Because of their interactive setting, however, just-in-time compilers often need to make a trade-off between *(verification+compilation) time* on the one hand, and *code quality* on the other—for example, by employing faster linear-scan register allocation instead of slower, but better, graph-coloring algorithms.

3 Proof-Carrying Code

Proof-Carrying Code (PCC) is a framework for ensuring that untrusted programs comply with a safety policy defined by the system where the programs will execute. Typical policies are type, memory, and control-flow safety, but in the framework originally described by Necula [2], any property of the program that can be expressed in first order logic constitutes a valid safety policy.

Upon reception of an untrusted program, the code consumer examines the code and computes a proof obligation for every operation that is potentially unsafe with respect to the safety policy. For instance, we might require a proof that a memory write lies within the bounds of a certain array allocated in the stack. Proof obligations are traditionally called *verification conditions*, or VCs.

The provider of an untrusted program must supply a proof for all the VCs. If a proof is missing, or the code consumer determines that it does not constitute a proof of the VC, the program is determined potentially unsafe and will not be executed. All this is done by examining the program and the proofs alone, without dependence on digital signatures or other mechanisms based on trust.

An important practical aspect of PCC is the size of proofs and the time spent in proof checking. It has been shown that proofs for Java type safety can be compressed to 12%–20% of the size of x86 machine code (a reduction of factor 30 with respect to a previous scheme), but unfortunately this increases the proof checking time by a factor of 3 [7]. While this work compresses the proofs, it does not reduce the number of facts that need to be proven.

However, in some mobile code contexts, 20% space overhead or long checking times are too much of an overhead. Our work aims at reducing both the size of proofs and the proof checking time by generating smaller VCs. For instance, by having a separate register file for basic types, no VCs are required to state the type of values contained in such registers.

Also, PCC has only been demonstrated in the context of machine code. Our

work carries over the same ideas to a target machine independent format, with all the advantages that this represents.

4 The Case For A Virtual Machine To Support Proof-Carrying Code

One of the reasons why proofs in PCC often are large is that the level of reasoning is very low, i.e. machine code level. At this level, registers and memory are untyped, and worse still, there is no differentiation between data values and address values (pointers). A large portion of each proof typically re-establishes typing of data, for example, distinguishing Integers from Booleans and from pointers.

Interestingly, current research on PCC (such as *Foundational Proof-Carrying Code* [8]) appears to be directed solely at reducing the size of the trusted computing base on the target platform. Unfortunately, this increases the size of the proofs that are required even further. We believe that it is much more promising to go the other way: by raising the semantic level of the language that proofs reason about, proofs can become much smaller. Facts that previously required confirmation by way of proof can then be handled by axioms. Our goal is to find such a higher semantic level that is at once effective at supporting proof-carrying code in this manner, and that can also be translated efficiently into highly performing native code on a variety of target platforms.

The framework we introduce in this paper works on this "higher semantic level". The enabling technology is a virtual machine that supports the concept of *tagged memory*, areas of memory that have an immutable tag that identifies the type of the data stored in this area. The virtual machine guarantees this property, i.e. the regular memory access instructions can access only locations that lie *within the data area* of a memory block—accesses are verified to lie within the range (beginning of block-end of block), which does not include the tag. Such an architecture greatly simplifies certain proofs: after the memory area has been initialized, instructions can no longer change the tag. Since at a higher level the tag relates to the (dynamic) type of a memory object, that implies that the type remains constant.

Using tagged memory, our framework creates a software defined layer at which proof-carrying code and dynamic translation can meet effectively. Hence, we also need to demonstrate the second half of the equation: how to make such a virtual machine efficiently implementable. The key issues here are type separation and referentially-safe encodings on the one hand (as previously demonstrated in the safeTSA project [3]), and an intricate memory addressing scheme on the other hand.

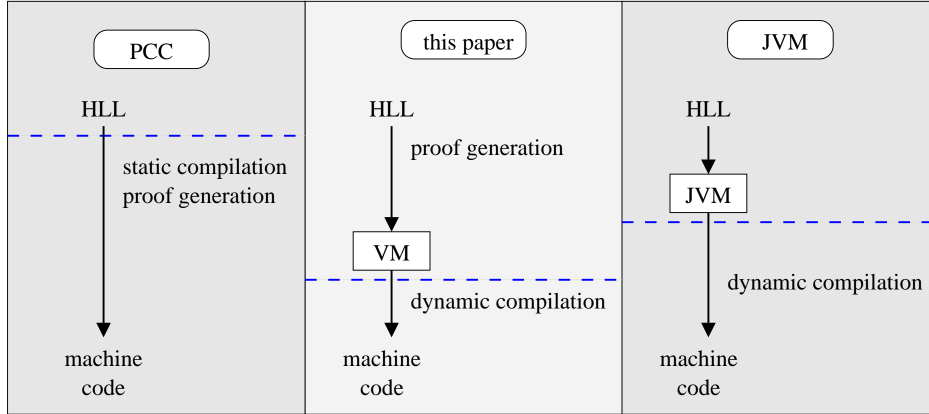

Fig. 2. Compiling high-level languages (HLL) to machine code in proof carrying code frameworks. Everything above the dashed lines is machine independent. Compared to current PCC implementations, our framework requires shorter proofs and is machine independent. Compared to current VMs, dynamic compilation is simpler because the semantic distance to actual target machines is smaller. Also, we are able to perform more optimizations ahead of time.

5 Architecture

The central element of our architecture is a division of concerns between the proof-carrying code mechanism and the virtual machine layer.

As described in the previous section, the virtual machine layer is designed to reduce the burden of proof by providing a number of inherently safe operations. As such safe operations often involve a runtime overhead, the safe vs. the non-safe properties of the VM have been carefully balanced. We chose to include only those inherently safe mechanisms that will have little or no overhead compared to equivalent non-safe operations. The proof-carrying code mechanism only has to provide proofs for the safety of the remaining parts of the VM architecture.

The rest of this section introduces different architectural aspects of our framework, namely typed register sets for scalar data types, object layout, accessing objects and arrays, dynamic guards, and runtime type identification.

5.1 Typed Register Sets for Scalar Data Types

Our VM architecture provides a finite number of scalar data types and complete separation between these types, except for explicit conversion operations. The VM has a separate register set (of unbounded size) for each of the basic types:

i_n: integer registers
b_n: boolean registers
p_n: pointer registers
a_n: address registers
...

The typed register sets in our architecture play a key role in reducing the complexity of proofs while maintaining safety at the same time. As the register sets are disjoint, type integrity of scalar values is enforced syntactically.

Some high-level types, such as enumerations, can be implemented as integers at the VM level. However, constraints on these types (e.g., no arithmetic allowed on the corresponding VM integer) cannot be enforced by the VM. Instead, such constraints must be enforced at the proof-carrying code level.

Being a register architecture, our VM can perform more ahead-of-time optimizations than stack-based code representations such as Java bytecode [1] or IL [9]. For example, constant folding, common subexpression elimination and copy propagation can all be performed ahead of time in our architecture.

5.2 Object Layout

In addition to the separation of basic type register files, the architecture of our VM aims to ease the task of providing proof of safety for memory access operations by offering an instruction set that guarantees memory integrity. There is a single instruction to allocate memory:

$$p_j = \texttt{new}(tag, i_k)$$

`new` allocates an array of objects of length i_k. Single objects are arrays with only one element. Allocated objects are tagged with the type that they represent. The VM layer ensures that the tag can only be written by the `new` instruction, but is immutable after object creation. Additionally, the VM offers instructions to read and check the object tag.

Objects are divided into two sections: one section for values and one section for pointers. The size of each of these sections is derived from the type information sent along with the mobile program. Figure 3 shows the Standard ML example program from [2]. The type representation generated from this program is shown in Figure 4. For each type we compute the aforementioned *type tag*, the *object layout* $(size_v, size_p)$, and the *structure* of the pointer section. $size_v$ (respectively $size_p$) is the size of the values (respectively pointers) section. The VM guarantees that pointers and values are not intermixed. The pair $(size_v, size_p)$ is also referred to as the *characterizing tuple* or *ctuple*. Finally,

```
datatype T = Int of int | Pair of int * int
fun sum (l : T list) =
 let
  fun foldr f nil a = a
    | foldr f (h::t) a = foldr f t (f(a, h))
 in
  foldr (fn (acc, Int i) => acc + i
          | (acc, Pair (i, j)) => acc + i + j)
        l 0
end
```

Fig. 3. The Standard ML example program from [2]. The program defines a union type `T` and a function `sum` that adds all the integers in a `T list`.

type	tag	layout	structure
`T list`	1	[0,8]	$\langle\{2,3\},\{1\}\rangle$
`Int of int`	2	[4,0]	$\langle\rangle$
`Pair of int*int`	3	[0,4]	$\langle\{4\}\rangle$
`int*int`	4	[8,0]	$\langle\rangle$

Fig. 4. Type information that is computed for the example program in Figure 3. High-level information is provided for the part of the object that stores references only. Access to the object area storing scalar values is permitted without strict type checks as this does not pose a safety risk.

for each entry in an object's pointer section the structure represents a set of possible run time types. For example, for the type `Pair of int*int` the only entry in the pointer section must have tag 4, that is be of type `int*int`.

To guarantee safe access to an object's contents, all memory accesses via a pointer must use the same type tag that was used to allocate the object. This must be enforced by the proof-carrying code layer. This mechanism can be understood as a rudimentary static type checking.

The value of type tags has no meaning for the VM layer, it is only interpreted by the PCC layer and is used to establish that an object has a certain layout without relying on a dynamic layout guard.

5.3 Accessing Objects

Separate instructions are provided to access values and pointers stored inside objects. To read and write value types from an object, either its *ctuple* or *tag* has to be specified. The *ctuple* allows the VM to properly access the object

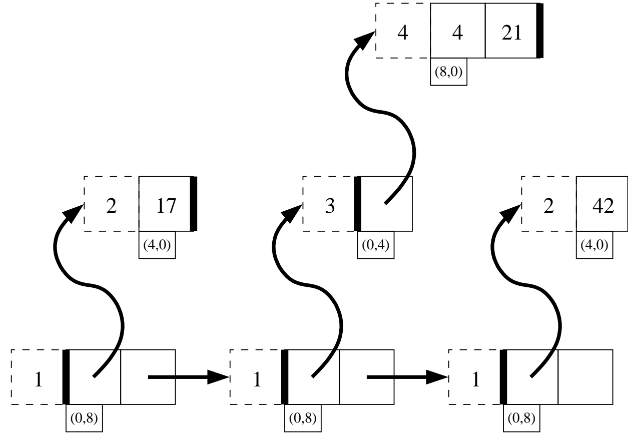

Fig. 5. Possible heap layout for the example program. The list elements (lowest line) contain a pointer to the actual data object and a pointer to the next list element. The data objects (middle line) either contain an integer or a pointer to an object (top line) containing two integers. All objects are tagged with their type (dashed box) as given in Figure 4. The bold vertical bars indicate the border between the value and the pointer section.

according to its layout and to (statically) check the specified offset. It is also permissible to specify a *tag* instead of a *ctuple*, in which case the PCC will substitute the *tag* with the corresponding *ctuple* before execution. There are read and write operations for the value register types:

$$
\begin{array}{l}
i_j = \texttt{iload}([size_v, size_p] \mid tag, p_k, \mathit{offset}) \\
\texttt{istore}([size_v, size_p] \mid tag, p_k, \mathit{offset}, i_l) \\
b_j = \texttt{bload}([size_v, size_p] \mid tag, p_k, \mathit{offset}) \\
\texttt{bstore}([size_v, size_p] \mid tag, p_k, \mathit{offset}, b_l) \\
\ldots
\end{array}
$$

Pointers are read and written using the pointer access instructions that also exist in two versions—one using the *layout* of objects, and one using the *type tag*:

$$
\begin{array}{l}
p_j = \texttt{pload}([size_v, size_p] \mid tag, p_k, \mathit{offset}) \\
\texttt{pstore}([size_v, size_p] \mid tag, p_k, \mathit{offset}, p_l)
\end{array}
$$

For the `pload` instruction that uses the *tag*, the PCC layer can derive the type of the object that p_j will point to by using the structure computed for the type tagged with *tag*. For example, using the information from Figure 4, for an instruction p_j `= pload(3,` p_k`, 0)`, the system can derive that p_j will point to an object of type `int*int`. For the tag version of the `pstore` instruction, the PCC layer can check that the object pointed to by p_l has a tag that complies with the structure of the object pointed to by p_k. For the instructions that use the *layout*, the code producer must provide additional proof that ensures these properties.

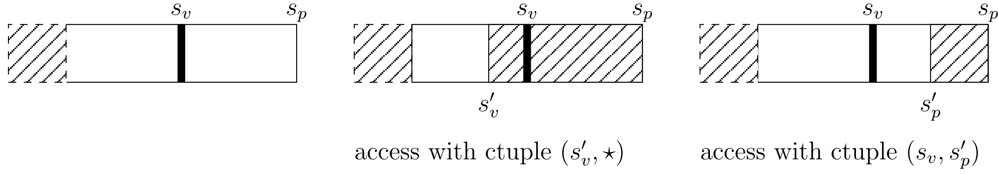

Fig. 6. Safe access to an object. For an object that was created with a ctuple (s_v, s_p), access to the shaded part of the object is prohibited by the PCC layer—the left part contains the type tag that can only be written during object creation, access to the right part is prohibited since the ctuple used for accessing the object is undefined for this region.

Additionally, the PCC layer has to ensure that the base pointer p_k was allocated with exactly the same allocation layout (expressed by $size_v$ and $size_p$), as specified for the load or store operation, and that the *offset* does not exceed $size_v$ or $size_p$ respectively. The proof-carrying code layer also has to ensure that pointer registers are always defined before their use.

Objects may also be accessed using a ctuple that is *considered safe* with respect to the ctuple that the object has been created with (c.f. Figure 6). A ctuple (s'_v, s'_p) is considered safe with respect to a ctuple (s_v, s_p) if $s'_v \leq s_v \wedge s'_p = \star$ or $s'_v = s_v \wedge s'_p \leq s_p$. An access based on (s'_v, s'_p) may only access those parts of the object that can be guaranteed to match the original ctuple. In the case of $(s'_v, \star)$ this means that only offsets less than s'_v into the value part of the object are allowed. Accordingly, (s_v, s'_p) allows to access the whole value part and offsets less than s'_p into the pointer part.

Using the described mechanisms, memory integrity can be implemented with little runtime overhead, while dynamically guaranteed type-safety is usually much more expensive in terms of runtime cost.

5.4 Accessing Arrays

Pointer registers always point to the beginning of arrays. To access other array members, there is an instruction taking a pointer p_k and an array index i_l and returning an address:

$$a_j = \texttt{adda}([size_v, size_p] \mid tag, p_k, i_l)$$

As for pointers, the proof-carrying code layer has to guarantee that address registers are defined before their use. Furthermore, proper memory layout has to be provided and ensured by the PCC layer. The `adda` instruction does not perform a mandatory array bounds check. If the proof-carrying code layer cannot provide a static proof of the index being within array bounds, a dynamic guard has to be inserted using the CHECKLEN instruction:

$\text{CHECKLEN}(p_k, i_l)$

When the virtual machine encounters a CHECKLEN instruction, it will perform a dynamic runtime check of the index i_l against the length of the array referenced by p_k. To implement this, the VM stores the array length of every allocated memory block in a memory block header. For the proof-carrying code layer, each occurrence of a CHECKLEN instruction establishes that at that point in the control flow the index in i_l will be appropriate for being used as array index for p_k, because the PCC layer can rely on the fact that this very runtime check will be performed by the VM layer.

Similar to the CHECKLEN instruction, which is used to check an index against the array length, the `getlen` instruction can be used to load the length of an array into a register:

$i_l = \texttt{getlen}(p_k)$

Unlike pointer registers, which point only to the start of arrays, address registers may point inside an allocated memory block. This has implications for garbage collection. Consider the following example:

1: $p_0 = \texttt{new}(tag, i_{length})$
2: $a_0 = \texttt{adda}([size_v, size_p], i_{offset})$
3: $p_0 = \texttt{null}$

An array of objects is allocated and a reference to the array is assigned into the pointer register p_0. `adda` is used to gain access to a member of the allocated array. A garbage collection cycle might be triggered after line 3, where the original reference to the array (in p_0) is no longer live, but a_0 is. a_0 is a *derived pointer* that keeps the array reachable [10]. If an accurate garbage collector is used, it either must be able to mark objects as reachable starting from derived pointers, or we must make sure that base pointers are considered part of the root set where the derived pointers are still live.

Since the regular memory access operations do not accept an address register as base address, the specialized address access instructions have to be used instead:

$i_j = \texttt{iloada}([size_v, size_p], a_k, offset)$
$\texttt{istorea}([size_v, size_p], a_k, offset, i_l)$
$b_j = \texttt{bloada}([size_v, size_p], a_k, offset)$
$\texttt{bstorea}([size_v, size_p], a_k, offset, b_l)$
$p_j = \texttt{ploada}([size_v, size_p], a_k, offset)$
$\texttt{pstorea}([size_v, size_p], a_k, offset, p_l)$

The rationale of this split is that the regular pointer-based memory access operations have to take the memory block header into account when calculat-

ing the target address while address access operations do not. Note that the address generation instruction `adda` cannot be applied to address registers, but only to pointer registers, and that memory access through address registers always requires to specify a concrete *ctuple*. Using address registers in conjunction with tags is undefined because only pointer registers, not address registers, allow access to the tag field, which is required by dynamic guard instructions as described next.

5.5 Dynamic Guards

The VM offers two more forms of dynamic guards in addition to CHECKLEN, which can be placed into the instruction stream if no static proof of certain properties can be provided. The PCC layer will treat these instructions as proof that the associated condition will be true at runtime, because it can rely on the virtual machine layer to actually perform these runtime checks.

CHECKNOTNULL ensures that a pointer register does not contain a `null` value:

$$\text{CHECKNOTNULL}(p_i) \quad \rightarrow_? \texttt{fail}$$

CHECKTAG verifies that a pointer points to a memory block with the specified memory layout or tag and fails otherwise.

$$\text{CHECKTAG}(p_i, [size_v, size_p] \mid tag) \quad \rightarrow_? \texttt{fail}$$

To support the CHECKTAG instruction, the VM stores the *tag* of every allocated memory block along with the array length in the memory block header. As the location of the memory block header is unknown for addresses in address registers, CHECKTAG only takes pointer registers as input. Instead, for address registers, CHECKTAG has to be applied to the base pointer used to generate the address in question.

In simple data flow scenarios, dynamic guards (CHECKLEN, CHECKNOTNULL, and CHECKTAG) can be avoided by supplying sufficient proof that the property in question was already true at an earlier point in the control flow. For more complex data flow scenarios, where for example a certain code location can be reached from several points in the program, it is up to the code producer to decide whether to use a static proof that a dynamic guard is not necessary or to actually emit a dynamic guard. The use of dynamic guards instead of static proofs is in particular beneficial in slow paths, because the proof size is minimized without significant runtime overhead.

The `iftag` instruction is used for runtime type identification. For the VM layer, `iftag` is a simple conditional branch depending on the value of the object tag. Along the taken-path of an `iftag` (and CHECKTAG) instruction, the proof-carrying code layer associates the pointer register with the supplied tag and allows certain assertions to pass, depending on high-level type information.

Using the type information computed for the example program (Figures 3 and 4), type `Pair of int*int` has the tag 3, the layout $[0, 4]$, and the structure $\langle\{4\}\rangle$. The type `int*int` has the tag 4. Using this information, the PCC is able to verify the code fragment given in Figure 7. When the PCC layer encounters the first CHECKTAG instruction, it assumes for the remainder of the basic block that the tag of `p0` is 3, because CHECKTAG will cause the VM layer to perform a dynamic check at this point that would fail if `p0` would point to an object with a different layout or tag. The second CHECKTAG instruction is a dynamic guard to ensure that the pointer read from the object of type `Pair of int*int` is in fact of type `int*int`, that is has tag 4. However, this dynamic guard is actually redundant because the tag for `p0` has already been established in this basic block and `p1` has been read at offset 0 from the pointer section of `p0`, which is known to have tag 4 according to the structure information $\langle\{4\}\rangle$ computed for type `Pair of int*int`.

For the PCC layer to be able to maintain tag associations across basic block boundaries, all predecessor basic blocks must either contain an explicit instruction that performs the check dynamically (guard) or asserts it, or the basic block itself must be annotated with a typemap that contains the proof needed.

To ensure safety, the VM layer and the PCC layer mutually depend on each other. The VM layer regards assertions accepted by the PCC layer as truth, and guarantees that all dynamic guards are performed at runtime. The PCC layer on the other hand relies on the VM to perform dynamic checks and takes the associated conditions for granted during verification.

```
...
CHECKTAG(p0, 3)
p1 = pload(3, p0, 0)
CHECKTAG(p1, 4)
...
```

Fig. 7. Code fragment that uses the type information from Figure 4. Using the CHECKTAG instruction, the PCC layer is able to prove that after the fragment `p1` points to an object with type `int*int`.

6 Proving Type-Safety in our VM

Our VM provides memory safety as well as type safety for primitive types such as `int`, `float` and `bool`. Given these, the proof burden of a code producer is greatly reduced in comparison to traditional PCC approaches. Since memory safety and primitive type safety are taken care of by the virtual machine, only proofs of type safety for non-primitive types are needed.

Consider, for example, the code for a simple factorial procedure, and the VM code for it as shown in Figure 8. Note that since this procedure only uses primitive types, *it is type-safe by construction.* The instruction set does not allow any type-unsafe operations (such as assigning integers to addresses). This further reduces the proof burden of the code producer, since type-safety proofs need only be produced for non-primitive types such as pointers, arrays, and records. Since substantial fractions of even object-oriented programs manipulate primitive types, this implies smaller proofs.

This is in sharp contrast to the Java bytecode instruction set. Java bytecode instructions do indicate the type of the operand being used (for example, `iload` for loading an integer, `fload` for loading a float, etc.). However, bytecode verification, as well as the technique of using stackmaps [11], must prove type-safety for bytecodes operating on values of primitive types as well. This is because of the Java virtual machine's stack-based memory model. The stack is a typeless entity, and once a datum is pushed onto the stack, information about its type is lost, and must be inferred again at the point that data is read from the stack. Thus, every load from the stack has to be proven type-

```
procedure fact (n : int) : int
begin
  f : int;
  f := 1;                        iconst 1, i0
  while (n > 0) do             loophead:
                                 iconst 0, i3
                                 bls i3, i1, b0
                                 brfalse b0, loopend
      f := f * n;                imul i0, i1, i0
      n := n - 1;                iconst 1, i3
                                 isub i1, i3, i1
  end;                           goto loophead
                               loopend:
  fact := f;                     % return value in i0
end
```

Fig. 8. High level language code and corresponding virtual machine representation of a simple factorial procedure.

safe, even for primitive types. We have shown that the complexity of Java bytecode verification can be exploited for denial-of-service attacks [12], and have proposed an alternative, efficient verification technique [13].

We measured the fraction of Java bytecodes that operate on primitive types (`int`, `float`, `double`, and `long`) in Section 3 (large scale applications) of the JavaGrande benchmark. Between 5% and 56% (with an average of 24%) of all bytecodes were of this type. For specJVM98, the range was between 22% and 43%, with an average of 29%. This is a crude measure of how much proof burden our virtual machine saves right away compared to the Java virtual machine.

So proofs are only needed for pointers and records. For every instruction that manipulates an address, we need to make sure that the resulting pointer

(1) points to the beginning of an array, or record, or value (e.g., pointers must not point into the middle of an integer)
(2) points to an object of the correct type (e.g., a pointer to an integer must not be allowed to point to a boolean)

For field accesses (using the `adda` instruction), condition 1 can be checked because offsets are known at compile time. For condition 2 and all pointers, our type-safety proofs take the form of *typemaps*. A typemap is a mapping from pointer and address registers to a set of their possible runtime type tags. Additionally, registers that have been proven to contain a `null` (respectively not `null`) value are annotated with *NULL* (respectively *NN*). At procedure entry, the typemap initially contains the declared types of the formal parameters and local variables. For each basic block in a procedure, the code producer can generate an annotation indicating what type tags a register will have at the basic block entry. From that annotation the proof-carrying code layer can derive the corresponding type tag at the basic block exit by sequentially inspecting the effect each instruction in the basic block on the register in question.

Before allowing the VM to execute code, the proof-carrying code layer has to make sure that the code is type-safe. For basic blocks that have been annotated as described above, this can be done as follows: [3]

- At the beginning of a basic block, set the derived typemap to the annotated typemap.
- Visit each instruction in the basic block in execution order, simulating its effect on the derived typemap and checking that the verification conditions for each instruction are discharged by the computed typemap.
- At the end of the block, for every successor S of this block, *match* the derived typemap with the annotated typemap at the beginning of S. This matching

[3] Note that this takes one linear pass over the code.

```
p: Int of int;
i: int;
...
if (i > 0) then
    p = new(Int of int);
else
    p = new(Int of int);
end;
...
i = *p;
```

Fig. 9. A program snippet that allocates memory with type `Int of int` along two different paths and assigns both to variable `p`. When this code fragment is translated to our VM representation, the register holding `p` will have to be annotated with additional layout information along each basic block associated with the *if* statement. This is done to prove that the object pointed to by this register has a certain unique layout after the *if* statement, namely $[4, 0]$ associated with type 2.

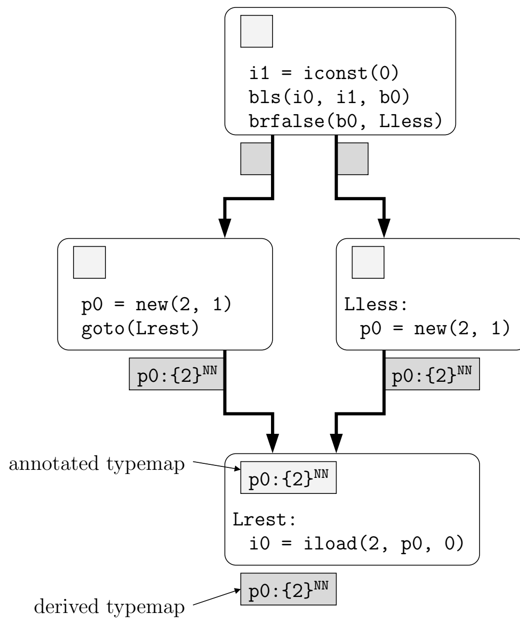

Fig. 10. Control flow graph annotated with typemaps for the example from Figure 9. The entry basic block is annotated with an empty typemap assuming that the code before this fragment does not contain operations on pointers or objects. Each outgoing edge is annotated with the typemap as derived by our checker.

procedure checks that every type in the successors annotated typemap is either more general than the corresponding type in the derived typemap or the successors typemap does not contain the variable in question.

An example is shown in Figure 9. The code allocates two memory blocks with identical layout along different paths. The blocks are both assigned to variable `p`. To prove that the register holding `p` has a unique layout after the

```
i0 = iconst(0)

Lagain:
  CHECKNOTNULL(P0)
  CHECKTAG(P0,1)
  p1 = pload(1,p0,0)
  CHECKNOTNULL(P1)
  iftag(p1,2,Linteger)

p0:{1}^NN, p1:{}^NN        p0:{1}^NN, p1:{2}^NN

  CHECKNOTNULL(P1)
  CHECKTAG(P1,3)
  p2 = pload(3,p1,0)
  CHECKNOTNULL(P2)
  CHECKTAG(P2,4)
  i1 = iload(4,p2,0)
  i2 = iload(4,p2,4)
  i1 = iadd(i1,i2)
  goto(Lsum)

p1:{3}^NN, p2:{4}^NN

Linteger:
  CHECKNOTNULL(P1)
  CHECKTAG(P1,2)
  i1 = iload(2,p1,0)

p1:{2}^NN

Lsum:
  CHECKNOTNULL(P0)
  CHECKTAG(P0,1)
  i0 = iadd(i1,i0)
  p0 = pload(1,p0,4)
  ifnull(p0,Lend)

p0:{1}^NN        p0:{}^NULL

  goto(Lagain)

Lend:
  return(i0)
```

Fig. 11. Necula's example from [2] using the mvm instruction set and runtime tag checks. For runtime type identification, a single `iftag` is used per iteration to distinguish between objects with type `Int of int` and `Pair of int*int`. Additionally, CHECKNOTNULL and CHECKTAG instructions are used to establish that all pointer access instructions are safe.

if statement, the code is annotated with typemaps as shown in Figure 10. Using the information from Figure 4, the type `Int of int` has the tag 2 and translates to a characterizing tuple of $[4, 0]$ in the VM, since it has 4 bytes in the data section and none in the pointer section. The derived typemap at the end of a block is compared with the annotated typemap of its successors. In this case, the two are the same, and thus checking succeeds. In the final basic block, the verification condition for the `iload` instruction is that `p` has been initialized, points to an object with type 2 and is not `null`. All these conditions can be discharged using the information in the annotated typemap.

Before we conclude this section, we revisit Necula's example from [2] as given in Figure 3. As noted above, the code producer in our framework has several choices of proving type safety for non-basic types. The one extreme is to enforce type safety purely dynamically by means of runtime checks. In this case, the code consumer would derive that before each pointer access the type of the pointer is checked dynamically. The resulting mvm code is shown in Figure 11. For each basic block, the code producer must provide enough proof that the

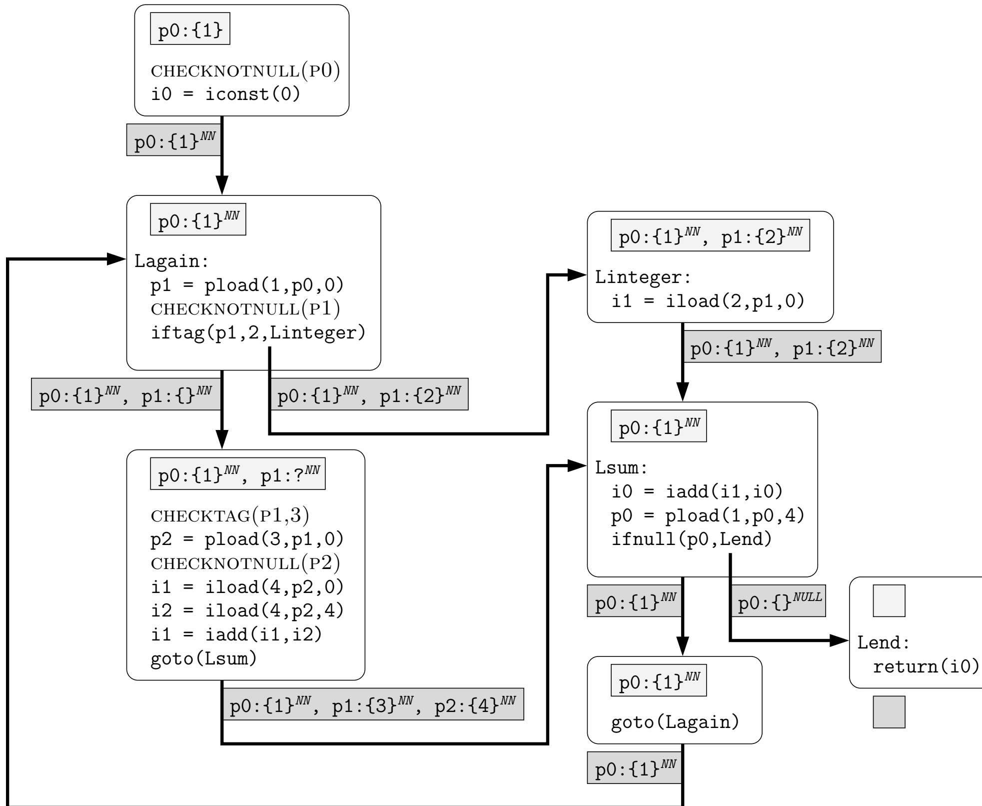

Fig. 12. Control flow graph annotated with typemaps for the example from Figure 11. As before, the light boxes are the annotated typemaps and the darker boxes are the typemaps derived by the checker at the code recipient. In contrast to 11 guards in the first solution (c.f. Figure 11), only 4 dynamic guards are needed.

code consumer can prove all pointer access operations to be safe. For example, before the first access to `p0`, the CHECKNOTNULL instruction ensures that the incoming object is not `null`, and the CHECKTAG instruction ensures that it has the correct type. As can be seen in Figure 11, the checker derives typemaps for each basic block, but since every successor block is annotated with an empty typemap, all tests for typemap matches succeed.

The other extreme would be to solely rely on static checks. However, this approach usually is not feasible since some tests can only be performed at runtime. Therefore, the code producer usually will have to choose a combination of static typemaps and dynamic checks. Figure 12 shows the control flow graph for the example from Figure 3, with the typemaps that have been annotated by the compiler and those that have been derived by the type-checker. In contrast to Figure 11, only 4 dynamic guards have been inserted by the compiler. In the basic block reached via the taken-path of the `iftag`, the PCC layer can derive that `p1` points to an object with type 2 simply by evaluating the arguments to the `iftag`. This is not possible for the path not taken. Instead, by inserting the CHECKTAG, the PCC layer is enabled to derive

that `p1` points to an object with type 3—otherwise, the execution would fail at this point. Additional guards are only needed to ensure that pointers read from objects are not `null`.

As can be seen in the annotated typemaps, the only fact that is known in the whole program is the type of `p0`. All other information is computed and proven locally but forgotten immediately, since it is not referenced in the annotated typemaps of successor basic blocks.

7 Related Work

There are several VM designs that use a low level instruction set similar to the one we are proposing. Keeping instruction sets close to real machines not only makes translation to real machine code fast and efficient but also makes it an attractive compilation target. However, the primary focus of most of these VMs is code optimizations rather than safe code.

The LLVM project [14] proposes to optimize the program not only at compile time but also during link and run time. To achieve this, they use a strongly-typed Static Single Assignment (SSA) based intermediate representation. Being SSA based and having type information makes optimizations fast and efficient. However, it supports some type unsafe cast operations for unsafe languages like C, hence for programs that use these operations few safety guarantees can be given.

The Dis virtual machine [15] provides an execution environment for the Inferno system. It is a CISC-like architecture with support for some high level data structures like lists and strings and operators to manipulate them. It allows for type unsafe operations and hence does not provide any type safety guarantees.

The Omniware system [16] was designed to provide an open system for producing and executing mobile code. The system was designed so that it is language and processor architecture neutral. It has an instruction set based on RISC architecture with several CISC-like functionalities. It uses Software Fault Isolation to provide module level memory safety. It does not provide any type safety and supports unsafe languages. Since the VM is very close to the real architecture it is able to achieve near to native speeds.

Typed Assembly Language (TAL) [17] is another framework for verifying the safety of a program for a low level representation. TAL uses the type system of the source language to prove the safety of the program. It achieves this by annotating the assembly code generated with high level type information available in the source code. These annotations are easily verifiable and are

used as proofs of safety. Before translation of the assembly code into a binary executable, the code is verified for safety using these annotations. An important point to note here is that there is no trust relationship between the compiler and the proof checker. The proof checker does not trust the compilation process or the annotations. At the time of compilation it will check for the correctness of the annotations.

Unlike PCC, which can use any safety policy expressible in first order logic, TAL only uses the typing rules of the programming language to express safety. This loss in generality of safety policies however leads to simpler, more compact and easy to generate proofs. It is not obvious how to automate the proof generation for policies more complex than memory and type safety.

Our system is in ways close to TAL as it also only supports type safety proofs. Our VM instruction set is strongly typed and has primitive types like `int` and `boolean`. Type safety for higher order types has to be proven using verifiable proofs. Since our VM provides a higher level of abstraction than machine assembly code and also provides some memory safety guarantees we claim that proofs will be shorter and even faster to verify. In case of segments of programs that use only primitive types, proofs are implicit in the instruction set and do not need any additional proofs.

SafeTSA [3] is a type-safe intermediate representation based on SSA. SafeTSA solves the problem of making SSA easily verifiable so that it can be used as a safe software transportation format. It does so by a combination of *type separation* and by introducing a *referentially safe* naming scheme for SSA values. As a consequence, the type and reference-safety of SafeTSA can be verified in linear time. Like the Java Virtual Machine itself, SafeTSA is tightly coupled to the Java type system and does not easily support languages with highly different type systems. SafeTSA is semantically further removed from the machine layer than the VM we have described in this article, and hence requires more substantial dynamic translation machinery at the target machine.

8 Conclusion and Outlook

We are exploring the design space of hybrid solutions between virtual machines and proof-carrying code. Our goal is to find the "sweet spot" that reconciles high execution performance and just-in-time compilation speed on the one hand, and small and efficient type-safety proofs on the other hand. In this article, we have reported on our first solution to populate this design space. Undoubtedly, there will be further candidates to follow. By combining virtual machines and proof-carrying code, we have been able to overcome the main limitations of these two techniques, namely cumbersome verification on the

one hand and lengthy proofs on the other. We are currently extending the set of properties that proofs in our framework can reason about by, e.g., values of index variables to avoid array bounds checks.

9 Acknowledgments

Parts of this effort are sponsored by the National Science Foundation (NSF) under grant CCR-TC-0209163, by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research (ONR) under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science foundation, or any other agency of the U.S. Government.

References

[1] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd Edition, The Java Series, Addison Wesley Longman, Inc., 1999.

[2] G. C. Necula, Proof-Carrying Code, in: POPL 1997, The ACM Symposium on Principles of Programming Languages, Paris, France, 1997, pp. 106–119.

[3] W. Amme, N. Dalton, M. Franz, J. V. Ronne, SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form, in: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, Snowbird, Utah, 2001, pp. 137–147, *SIGPLAN Notices,* 36(5), May 2001.

[4] V. Haldar, C. H. Stork, M. Franz, The Source is the Proof, in: The 2002 New Security Paradigms Workshop, ACM SIGSAC, ACM Press, Virginia Beach, VA, USA, 2002.

[5] W. Amme, N. Dalton, P. H. Fröhlich, V. Haldar, P. S. Housel, J. von Ronne, C. H. Stork, S. Zhenochin, M. Franz, Project transPROse : Reconciling Mobile-Code Security with Execution Efficiency, in: DARPA Information Survivability Conference and Exposition, 2001.

[6] R. F. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer-Verlag, 2001.

[7] G. C. Necula, S. P. Rahul, Oracle-Based Checking of Untrusted Software, in: POPL 2001, The ACM Symposium on Principles of Programming Languages, London, United Kingdom, 2001, pp. 142–154, *SIGPLAN Notices,* 36(3), March 2001.

[8] A. Appel, Foundational Proof-Carrying Code, in: 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), IEEE, 2001, pp. 247–258.

[9] ISO/IEC 23271, Common Language Infrastructure (CLI), Partition III, CIL Instruction Set (Dec 2002).

[10] A. Diwan, J. E. B. Moss, R. L. Hudson, Compiler Support for Garbage Collection in a Statically Typed Language, in: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI), 1992, pp. 273–282, *SIGPLAN Notices* 27(7), July 1992.

[11] Sun Microsystems Inc., Connected, Limited Device Configuration (Apr 2000).

[12] A. Gal, C. W. Probst, M. Franz, A Denial of Service Attack on the Java Bytecode Verifier, Tech. Rep. 03-23, University of California, Irvine, School of Information and Computer Science (November 2003).

[13] A. Gal, C. W. Probst, M. Franz, Proofing: An Efficient and Safe Alternative to Mobile-Code Verification, Tech. Rep. 03-24, University of California, Irvine, School of Information and Computer Science (November 2003).

[14] C. Lattner, LLVM: An Infrastructure for Multi-Stage Optimization, Master's thesis, University of Illinois, Urbana Champaign, Urbana, Illinois (2000).

[15] P. Winterbottom, R. Pike, The Design of the Inferno Virtual Machine, in: Hot Chips IX: Stanford University, Stanford, California, August 24–26, 1997, IEEE Computer Society Press, 1997.

[16] S. L. A. Adl-Tabatabai, G. Langdale, R. Wahbe, Efficient and Language-independent Mobile Programs, in: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, 1996, pp. 128–136.

[17] K. C. Greg Morrisett, David Walker, N. Glew, From System F to Typed Assembly Language, in: POPL 1998, The ACM Symposium on Principles of Programming Languages, San Diego, CA, USA, 1998, pp. 85–97.