A Portable Virtual Machine Target For Proof-Carrying Code

Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermín Reig, Ning Wang
{franz,dchandra,gal,vhaldar,reig,wangn}@uci.edu
Department of Computer Science
University of California
Irvine, CA 92697-3425

ABSTRACT

Virtual Machines (VMs) and Proof-Carrying Code (PCC) are two techniques that have been used independently to provide safety for (mobile) code. 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 sizes of the proofs and the fact that the certified code is no longer machine-independent. In this paper, we describe work in progress on combining these approaches. 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.

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), we *verify* 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* [10], *proof-carrying code* [11], and *inherently safe code formats* [3, 7, 2].

In virtual machines 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 example, virtual machines have general *GOTO* instructions but not all possible control flows are actually legal[2]. 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 dataflow 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 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 paper is structured as follows: First, we discuss the Java

[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 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, don't easily map back onto the control structures of the source language.

IVME'03, June 12, 2003, San Diego, California, USA.

Virtual Machine, not only because it is a good representative of the VM genre, 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 architecture is still being refined and we are merely presenting the current state of a work in progress. This section also contains several examples. Section 6 expands on the proof-carrying code aspects of our work. Section 7 presents related work. Finally, we give an outlook to future work and conclude the paper.

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. In addition, it has been shown that the rules for bytecode verification don't 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[13].

Further, due to the need to verify the code's safety upon arrival at the target machine, and also due to the specific semantics of 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. Or take common subexpression elimination: a compiler generating Java bytecode could in principle perform common subexpression elimination and store the resulting expressions in additional, compiler-created local variables, but this approach is incomplete because it cannot eliminate common address calculations (for arrays etc.)—since verification requires preserving the language abstractions, such optimizations for the JVM can be performed only *after* verification (i.e., on the target machine).

The problem here is that just-in-time compilation occurs while interactive users are 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 one side, 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 described by Necula [11], 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 emits a proof obligation for all operations that are 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 factor of 30 reduction with respect to a previous scheme), but unfortunately this increases the proof checking time by a factor of 3 [12]. Notice that this work compresses the proofs, but does not reduce the number of facts that need to be proved.

In some mobile code contexts, 20% space overhead or large checking times are considered too high. Our work aims at reducing the size of proofs and the proof checking time by generating smaller VCs. For instance, by having a separate register file for booleans, 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 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 PCC often has very large proofs is because 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* [4]) appears to be directed solely at reducing the size of the trusted computing base on the target platform. Unfortunately, this increases the volume 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 the proofs reason about, the proofs can become much smaller. Facts that previously required confirmation by way of proof now can 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.

As an example of what we mean by "higher semantic level", imagine a virtual machine that supports the concept of *tagged memory*, areas of memory that have a tag stored at an offset from the pointer that is unreachable via the regular memory access instructions. 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 doesn't include the tag. Conversely, access to the tag area requires one of two *privileged instructions*, only one of which reads and the other of which writes the tag value. Such an architecture greatly simplifies certain proofs: any fragment of code that doesn't use the privileged "write-to-tag" instruction cannot have changed the tag. Since at the higher level the tag relates to the (dynamic) type of a memory object, that implies that the type has remained constant.

Our aim is to create 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 here is our use of type separation and referentially-safe encodings on one hand (as previously demonstrated in the safeTSA project [3]), and of an intricate memory addressing scheme on the other hand.

5. ARCHITECTURE

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

The virtual machine layer is designed in a way 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 to be carefully balanced. We choose to include only those safe-by-construction mechanisms which we expect will have no or very little overhead compared to equivalent non-safe operations. The proof-carrying code mechanism only has to provide proof for the safety of the remaining parts of the VM architecture.

Our VM architecture provides a finite number of scalar data types and complete type 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 the 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. 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 allows to apply more ahead-of-time optimizations than stack-based code representations like Java bytecode [10] or IL [8]. For example, constant folding, common subexpression elimination and copy propagation are all performed ahead of time in our architecture.

The architecture of our VM aims to ease the task of providing proof of safety for memory access operations by offering an instruction set which guarantees memory integrity. There is a single instruction to allocate memory:

$$p_j = new(size_v, size_p, i_k)$$

new allocates an array of objects of length i_k. (Single objects are arrays with a single element.) Allocated objects are divided into two sections: one section for values and one section for pointers. $size_v$ (resp. $size_p$) is the size of the values (resp. 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*.

To guarantee safe access to an object's contents, all memory accesses via a pointer must use the same *ctuple* 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.

Separate instructions are provided to access values and pointers stored inside objects. There are read and write operations for the value register types:

$$\begin{gathered} i_j = iload(size_v, size_p, p_k, \mathit{offset}) \\ istore(size_v, size_p, p_k, \mathit{offset}, i_l) \\ b_j = bload(size_v, size_p, p_k, \mathit{offset}) \\ bstore(size_v, size_p, p_k, \mathit{offset}, b_l) \end{gathered}$$

Pointers are read and written using the pointer accesss instructions:

$$\begin{gathered} p_j = pload(size_v, size_p, p_k, \mathit{offset}) \\ pstore(size_v, size_p, p_k, \mathit{offset}, p_l) \end{gathered}$$

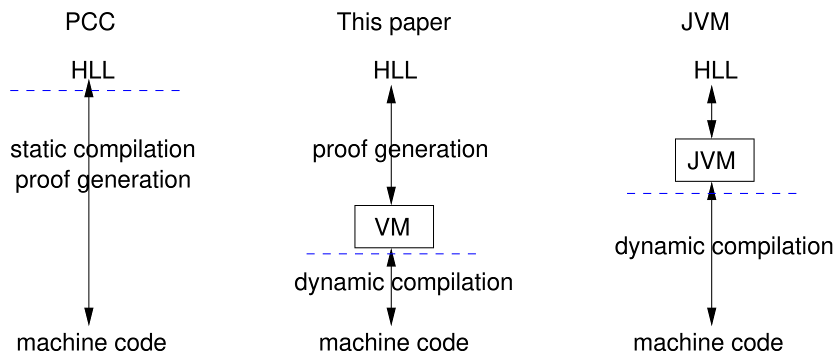

Figure 1: Everything above the dashed line is machine independent. Compared to current PCC implementations, our framework requires shorter proofs and is machine independent. Compared to current VMs, dynamic compilation is simpler because the semantic distance to actual target machines is smaller. Also, we are able to perform more optimizations ahead of time.

The verification 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.

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

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:

$$a_j = adda(size_v, size_p, p_k, i_l)$$

As for all memory access operations, 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.

$$checklen(p_k, i_l)$$

checklen raises an exception if i_l exceeds the length of the array referred to by p_k. To implement this, the VM stores the array length of every allocated memory block in a memory block header. The *getlen* instruction loads the length of an array into a register.

$$i_l = getlen(p_k)$$

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

$$\begin{array}{c} 1\text{: } p_0 = new(size_v, size_p, i_{length}) \\ 2\text{: } a_0 = adda(size_v, size_p, i_{offset}) \\ 3\text{: } p_0 = null \end{array}$$

An array of object 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 could be triggered after line 3, when the original reference to the array (in p_0) is not live but a_0 is. a_0 is a *derived pointer* that keeps the array reachable [5]. 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.

As for pointers, the proof-carrying code layer has to guarantee that address registers are defined before their use.

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

$$\begin{array}{c} i_j = iloada(size_v, size_p, a_k, offset) \\ istorea(size_v, size_p, a_k, offset, i_l) \\ b_j = bloada(size_v, size_p, a_k, offset) \\ bstorea(size_v, size_p, a_k, offset, b_l) \\ a_j = ploada(size_v, size_p, a_k, offset) \\ pstorea(size_v, size_p, a_k, offset, p_l) \end{array}$$

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

The VM offers, in addition to *checklen*, two more forms of dynamic guards which can be placed into the instruction stream if no static proof of certain properties can be provided. The PCC layer can treat the existence of these instructions in the instruction stream as proof that the associated condition will be true at runtime (or an exception will be raised).

checknotnull raises an exception if a pointer register contains the value *null*.

$$checknotnull(p_i) \quad \rightarrow_? exception$$

checklayout verifies that a pointer points to a memory block with the specified memory layout and raises an exception otherwise.

$$checklayout(p_i, size_v, size_p) \quad \rightarrow_? exception$$

To support the *checklayout* instruction, the VM stores the *ctuple* 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, *checklayout* only takes pointer registers as input. Instead, *checklayout* has to be applied to the base pointer previously used to generate the address in question.

6. PROOFS FOR OUR VM

Our VM provides the following guarantees: memory safety, and type-safety for the primitive types of int, floats and bools.

Given this, the proof burden of a code producer is greatly reduced. 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.

```
procedure fact (n : integer) : integer
begin
        f : integer;
        f := 1;
        while (n > 0) do
                f := f * n;
                n := n - 1;
        end;
        fact := f;
end
```

```
iconst 1, i3
imov i3, i2
loophead :
iconst 0, i4
bls i4, i1, b0
brfalse b0, loopend
imul i2, i1, i5
imov i5, i2
iconst 1, i6
isub i1, i6, i7
imov i7, i1
goto loophead
loopend :
imov i2, i0
```

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 booleans). 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, we expect 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[14], 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-safe, even for primitive types.

We measured the fraction of Java bytecodes that operate on primitive types (ints, floats, doubles and longs) 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 the beginning of an array, or record, or value (e.g. we must not have pointers that point into the middle of an int)

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 simply a mapping from pointers to their types. At procedure entry, these are simply the declared types of the formal parameters and local variables. During code generation, these are emitted for the beginning of every basic block.

Before letting the VM execute code, our type-checker makes sure the code is type-safe. Given that the code has been annotated with typemaps as explained above, the checker does the following:

- At beginning of a basic block, set derived typemap to annotated typemap

- Iterate through each instruction in the basic block, simulating its effect on the derived typemap

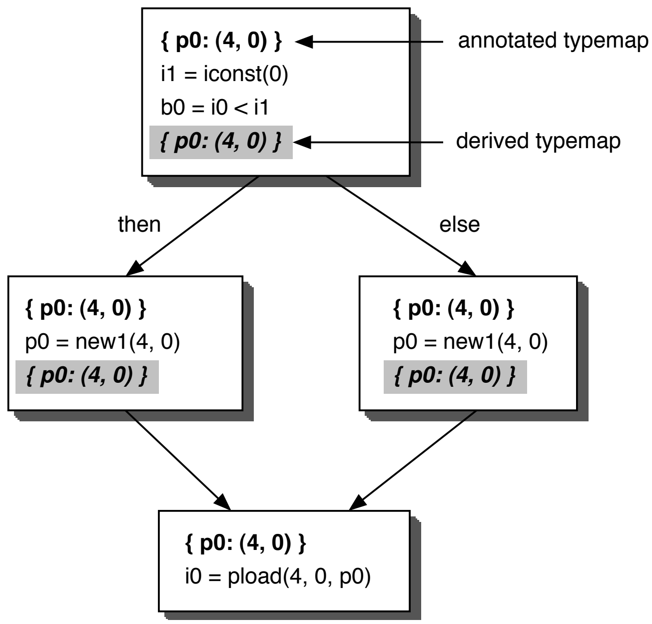

Figure 2: Simple example illustrating the use of typemaps for proofs of type safety. In this example, we are using the instruction "new1" as a shorthand for "new(*, *, 1)" to achieve syntactic similarity with the typemap.

- At the end of the block, for every successor B1 of this block, *match* the derived typemap with the annotated typemap at the beginning of B1. This matching procedure simply checks that every type in the successors typemap is more general than the corresponding type in the derived typemap.

Note that this takes one linear pass over the code.

For a very simple example consider the following code snippet.

```
p: pointer int
i: int

if (i > 0) then
        p = new(1, sizeof(int));
else
        p = new(1, sizeof(int));
end;
```

Figure 2 shows the corresponding VM code with annotated and derived typemaps. The type `int` 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 at the beginning of all its successors. In this case, the two are the same, and thus, checking succeeds.

7. RELATED WORK

There are several virtual machine(VM) designs that use a low level instruction set similar the one we are proposing. Keeping instruction sets close to real machine 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 [9] 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 SSA based intermediate representation(IR). Being SSA based and having type information makes optimizations fast and efficient. However, it supports some type unsafe cast operations for unsafe languages and hence does not provide any type safety. The project focuses only on the IR and does not have any runtime available.

Tao's Intent [15], a commercial system, focuses on binary portability mainly of multimedia/games application. The VM at the heart of the system is very simple and low level. Even though it aims at binary portability it does not provide any type safety. This is partly because they wanted to support legacy code written in C/C++ and partly for the reasons of performance, both being important factors for multimedia applications.

The Dis virtual machine [16] provides an execution environment for the Inferno system. It is a CISC like architecture with support for some high level data structures like list 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 [1]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 (SFI) 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)[6] 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 quickly 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 quickly 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 proved using verifiable proofs. Since our VM provides a higher level of abstraction than a machine assembly code and also provides some memory safety guarantees we claim that the proofs will be shorter and even faster to verify. In case of segments of programs that use only primitive types the proofs are implicit in the instruction set and doesn't need any additional proofs.

SafeTSA[3] is a type-safe intermediate representation based on Static Single Assignment form (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 paper, and hence requires a 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 one hand, and small and efficient type-safety proofs on the other hand. In this paper, we have reported on the design of our first solution to populate this design space; undoubtedly, there will be further candidates to follow. Preliminary evidence suggests that hybrid solutions have advantages over single-mode VM or PCC approaches.

9. ACKNOWLEDGEMENTS

Parts of this effort are sponsored by the National Science Foundation under grants CCR-TC-0209163, by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research 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 (DARPA), the National Science foundation (NSF), or any other agency of the U.S. Government.

10. REFERENCES

[1] S. Lucco A. Adl-Tabatabai, G. Langdale and R. Wahbe. Efficient and language independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and ImplementationProgramming Language Design and Implementation*, pages 128–136, May 1996.

[2] Wolfram Amme, Niall Dalton, Peter H. Frohlich, Vivek Haldar, Peter S. Housel, Jeffrey von Ronn, Christian H. Stork, Sergiy Zhenochin, and Michael Franz. Project transprose : Reconciling mobile-code security with execution efficiency. In *DARPA Information Survivability Conference and Exposition*, June 2001.

[3] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael 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, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[4] A. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Washington - Brussels - Tokyo, 2001. IEEE.

[5] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 273–282, 17–19 June 1992. *SIGPLAN Notices* 27(7), July 1992.

[6] Karl Crary Greg Morrisett, David Walker and Neal Glew. From system f to typed assembly language. In *"25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages"*, pages 85–97, San Diego, CA, USA, "January" 1998.

[7] Vivek Haldar, Christian H. Stork, and Michael Franz. The source is the proof. In *New Security Paradigms Workshop*, Sep 2002.

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

[9] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois, Urbana Champaign, Urbana, Illinois, 2000.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[11] George C. Necula. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.

[12] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, London, United Kingdom, January 17–19, 2001. *SIGPLAN Notices,* 36(3), March 2001.

[13] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001.

[14] Sun Microsystems Inc. Connected, Limited Device Configuration, Apr 2000.

[15] Introduction To Intent Technology. Available at `http://withintent.biz/aux-files/Introduction_to_intent_Technology.pdf`, 2002.

[16] Phil Winterbottom and Robert Pike. The Design of the Inferno virtual machine. *Hot Chips 9 symposium*, August 1997.