An Untrusted Verifier for Typed Assembly Language

Adam Chlipala (adamc@cs.berkeley.edu)

1. INTRODUCTION

Software downloaded onto computers and executed without user oversight or intervention, otherwise known as mobile code, is playing an increasingly prominent role in today's computing infrastructure. Java applets and ActiveX components are well known to the general public. Software components are routinely downloaded to and executed by cellular phones, smart cards, and other devices that consumers generally don't think of as sophisticated computers. The seemingly innocuous nature of mobile code for these applications opens up opportunities for exploitation by malicious software creators.

The traditional technique for preventing such abuses involves running untrusted code in a sandbox, inserting many dynamic run-time tests, and other methods that can add significant execution overhead. Newer methods rely more strongly on static analysis of code to prove absence of undesirable behavior. The Java Virtual Machine (JVM) format [4] and the Microsoft .NET Common Language Infrastructure (CLI) [1] are both designed to allow static verification of basic properties, including memory safety. However, they depend on the inclusion of many very high level instructions, including those for object-oriented features, different types of function calls, and more. Thus, while these formats are designed to be easy targets for many high level programming languages, it is common that languages sufficiently different from those original considered will have sufficiently different features to make the JVM and CLI formats poor targets.

The Open Verifier project seeks to develop a more flexible and trustworthy alternative to these infrastructures. It is based around the idea of minimizing the set of abstractions built into the software verification process. Instead of using bytecode formats with many high-level instructions, it verifies programs written in standard assembly languages. The job of proving memory safety of these programs is delegated to untrusted modules, which produce proofs that are checked by a small trusted core. Each compilation strategy for each high level language can have a separate module of this kind. This frees language designers and compiler implementors to consider many different implementation strategies without being constrained by language specific features of a target format or requiring constant addition of new instructions to that format. The Open Verifier architecture's design also reduces the amount of code that a mobile code user must trust.

I am currently developing an untrusted Open Verifier module to enable checking of programs compiled from Typed Assembly Language (TAL) [5]. TAL is similar to the common language target formats described above. However, it uses a sophisticated type system to remove most need for instructions at a higher level than those generally provided by real processors. Most TAL instructions are real machine instructions augmented with typing information.

The status of TAL as possibly the most expressive language of its kind makes producing a verification module for it a good test of the flexibility of the Open Verifier system. Here I present a fairly straightforward encoding of its type system in the mostly first-order logic system used by the Open Verifier. While the verifier that uses this encoding is not yet complete, the chances that the final product will integrate well with the Open Verifier seem promising.

2. RELATED WORK

2.1 Prior work on safety of low level code

Proof-carrying code (PCC) [7] is the inspiration for the Open Verifier architecture. PCC introduced the idea of a certifying compiler, which can produce a safety proof for each machine code program it produces. The proof system a compiler uses is custom tailored to the high level language it is compiling. While this simplifies the compiler's burden, it also requires much repeated effort in producing proof checking infrastructures for different languages and proving the proof systems sound. There is also no common ground on which an end user can base trust of different PCC systems. This last point is a serious weakness for using traditional PCC as a basis for a flexible and safe mobile code platform.

Foundational PCC (FPCC) [2] attempts to repair this defect. An FPCC system creates a PCC infrastructure "from the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system" [2]. Properties of type systems that traditional PCC takes as axioms are defined explicitly in higher-order logic based only on underlying machine properties. The needed properties can be proved, and the the proofs can be checked by a minimal higher-order logic proof checker. However, the expressivity of higher-order logic does not come without a price. The proof of soundness for each type system under consideration is usually quite involved. Relations indexed by the number of execution steps [3] for which particular code is known to run safely "infect" the proofs to a degree that can be confusing for those who might want to implement FPCC for their languages' type systems. The proofs end up containing many aspects that don't seem central to why the corresponding type systems imply memory safety.

Typed Assembly Language (TAL) [5] takes a different ap-

proach to remedying the same problem. Instead of removing a specific type system and its properties from the trusted computing base, a type system expressive enough to be a good target for a wide range of high level languages is used. A TAL for a particular processor architecture modifies that architecture's assembly language by adding typing information to the existing instructions and adding a few extra high level instructions. For example, the first release of x86 TAL added 8 high level instructions that require concrete execution while not corresponding to real x86 instructions. This is less than the number of instructions the Microsoft CLI features for different kinds of function calls alone. However, these few instructions still impose some limitations on possible runtime optimizations. For example, the built-in array type must be accessed through checked subscript read and update operations, removing opportunities for eliding unnecessary bounds checks. The type system is not expressive enough to support an alternate implementation of arrays with the same performance characteristics. Also, it seems unlikely that language designers will never invent new features that aren't adequately handled by a single monolithic type system.

2.2 The Open Verifier

The Open Verifier architecture is an alternative to FPCC as a successor to PCC. They have similar goals but achieve them through different means and with different emphases. Building on work on untrusted proof rules [8] and verification condition generators [9], the Open Verifier architecture, like FPCC, avoids building program semantics above the machine code level into its trusted base. The two systems diverge when it comes to choosing which additional pieces of a complete program verifier to move into the trusted base.

Besides the unavoidable semantic model of machine code execution, the Open Verifier includes a simple engine for traversing the reachable states of a program. Checking all possible states is an inherent requirement of verification of safety properties, so this does not impose undue constraints.

The exact nature of the descriptions of individual states is left up to an untrusted *extension* module that is specific to the high level language and compilation strategy used to produce the program being checked. An extension writer chooses an encoding of possible machine states in first-order logic, where the ground theory only includes notions related to the underlying machine architecture and execution environment.

To verify a program's safety, the trusted core of the Open Verifier performs a traversal of the program's reachable states. A trusted strongest postcondition generator produces exact first-order logic characterizations of the different machine states that may follow a given one, while the current extension produces its own first-order logic characterizations of successor states. The extension's state descriptions will generally be less detailed than the concrete state descriptions. For instance, while the built-in postcondition generator may track the exact values of all machine registers, an extension may choose to track only the types of those values. It is the extension's ability to drop information while keeping enough to prove the needed safety properties that makes verification practical.

Of course, verification is not sound if the Open Verifier allows the extension to declare successor states that have no relation with actual machine behavior. It is important that an extension be able to prove that the abstract successors it declares somehow cover all possible concrete successor states.

In a system like FPCC, this could be proved with standard higher-order logic proof rules. The Open Verifier attempts to keep the job of the extension writer simple by sticking to first-order logic. To handle this one case which seems to call for higher-order reasoning, a single new proof rule is introduced, the *coverage rule*. It is based on fairly intuitive notions of when any machine state that realizes any member of a set of abstract states must also be realized by some machine state realizing an abstract state from another set. This rule is used to show that any concrete successor state of a particular program state is covered by one or more of the abstract successors chosen by the extension.

It is possible to prove the soundness of the coverage rule and the Open Verifier architecture in general. The greatest complexity arises in making the notion of coverage general enough to handle the wide variety of ways programs can use code addresses. The idea of a *progress continuation* is introduced to allow state descriptions to express higher-order facts, in the form of formulas such that any machine state satisfying one of them is known to be safe. The proof uses indexed relations much like those used for FPCC soundness proofs. However, the soundness proof only needs to be done once, not once per type system.

For the purposes of this paper, the details of coverage are not important. The TAL type system is expressive enough that the needed uses of coverage can be folded into a single construct, the idea of one global progress continuation that is essentially a first-order formula such that any machine state satisfying it will execute safely from that point on. At the beginning of verification, the extension must prove that every machine state satisfying this predicate has in fact been queued to be verified, using the coverage rule. This will be described in more detail in later sections.

3. AN OVERVIEW OF X86 TAL

Here I will present a fragment of the parts of TAL for Intel x86 processors that are interesting in the context of this project. In particular, an extension for a language supporting first class functions or parametric polymorphism has not yet been produced for the Open Verifier. TAL is well suited as a target for such languages, as shown in the transformation from System F (the polymorphic typed lambda calculus) to TAL of [6]. Therefore, I will consider a TALx86 fragment that brings out the issues involved in supporting these features.

3.1 Grammar

Figure 1 gives the grammar for this fragment. None of the features under consideration require the use of any of the high level TALx86 instructions not corresponding to standard x86 assembly instructions. This means that, for this fragment, TAL simply provides typing information on top of a traditional assembly program, so I will not give the details of the instructions and machine values involved. The typing information can be thought of as a *register typing* for each instruction in an assembly program, where a register typing is a mapping from each register in a subset of the x86 registers to a TALx86 *constructors*; as well as assignments of types to selected code labels within a program, such as function entry points. Constructors encompass both types

Type variables	α		
Kinds	k	::=	$\texttt{k4byte} \mid \texttt{kstack} \mid k_1 \rightarrow k_2$
Type contexts	Γ	::=	$\cdot \mid \Gamma, \alpha :: k \mid \Gamma, \alpha = \tau$
Registers	r	::=	$\texttt{eax} \mid \texttt{esp} \mid ...$
Register typings	T	::=	$\{r_1 : \tau_1, ..., r_n : \tau_n\}$
Constructors	τ	::=	$\texttt{word} \mid \texttt{code}\ T \mid \alpha \mid \forall \alpha :: k.\tau \mid \texttt{nil} \mid \texttt{cons}(\tau_1, \tau_2) \mid \texttt{stackptr}(\tau)$
Expressions	e, m		
Code labels	L		
Register values	V	::=	$\{r_1 = e_1, ..., r_n = e_n\}$

Figure 1: Basic x86 TAL grammar

$$\forall \alpha :: \texttt{kstack.code}\ \{\texttt{esp} : \texttt{stackptr}(\texttt{cons}(\texttt{code}\ \{\texttt{eax} : \texttt{word}, \texttt{esp} : \texttt{stackptr}(\texttt{cons}(\texttt{word}, \alpha))\}, \texttt{cons}(\texttt{word}, \alpha)))\}$$

Figure 2: A TALx86 type for a pointer to the entry point of a C function with prototype `int f(int)`, using standard x86 calling conventions

describing runtime values and "type-level terms" that can be used to compute types programmatically. Both actual machine values and values representing entire memory states will be named with *expressions*.

To handle polymorphism, *type variables* and *type contexts* are introduced. Type contexts map unique type variables to TALx86 types and *kinds*. A kind is a grouping of types. For instance, there are types whose values are all word-sized and types whose values are all multi-word stacks in memory. These categories of types are given different kinds, `k4byte` and `kstack`. When a type context maps a type variable to a type, it indicates that the type variable stands for the given type. When a type variable is mapped to a kind, it indicates that the variable stands for some type of that kind. The latter case will be used for varieties of parametric polymorphism that are sensitive to such properties as the amount of storage needed to hold values of a universally bound type variable.

3.2 Judgments

The type system is made up of three basic judgments:

$\Gamma \vdash \tau :: k$ means that constructor τ has kind k in context Γ. Γ provides assumptions about type variables bound in τ in the usual way.

$\Gamma \vdash_m e : \tau$ means that expression e has type τ in context Γ with memory contents m.

$\Gamma \vdash_m V : T$ means that register values V are compatible with register types T in context Γ with memory contents m. This means that for each $r_i : \tau_i$ appearing in T, $r_i = e_i$ such that $\Gamma \vdash_m e_i : \tau_i$ must appear in V.

3.3 Constructors

There are four basic varieties of constructors, each based on a notion of type that might be applied informally in describing the execution of an x86 assembly language program.

3.3.1 Primitive types

$\Gamma \vdash \texttt{word} :: \texttt{k4byte}$. `word`'s represent any 4-byte quantities.

3.3.2 Code types

$\Gamma \vdash \texttt{code}\ \{r_1 : \tau_1, ..., r_n : \tau_n\} :: \texttt{k4byte}$. Such a code type is the type of pointers to code blocks that are safe whenever the values of registers $r_1, ..., r_n$ have the specified types.

3.3.3 Stack types

`nil` is the type of an empty stack. ($\Gamma \vdash \texttt{nil} :: \texttt{kstack}$) When $\Gamma \vdash \tau_1 :: \texttt{k4byte}$ and $\Gamma \vdash \tau_2 :: \texttt{kstack}$, $\Gamma \vdash \texttt{cons}(\tau_1, \tau_2) :: \texttt{kstack}$. This is the type of stacks with values of type τ_1 on top and stacks of type τ_2 below them. When $\Gamma \vdash \tau :: \texttt{kstack}$, $\Gamma \vdash \texttt{stackptr}(\tau) :: \texttt{k4byte}$. This is the type of pointers to stacks of type τ.

3.3.4 Universal types

When $\Gamma, \alpha :: k_1 \vdash \tau :: k_2$, $\Gamma \vdash \forall \alpha :: k_1.\tau :: k_1 \rightarrow k_2$. This is the type for universal quantification over a type variable of a particular kind.

3.4 Compilation of functions to TAL

TAL doesn't have any notion of functions. The usual function call-return idiom is captured by giving continuation passing style typings to the actual register values involved during a function call. For instance, using the standard x86 calling conventions, a function that takes a single integer parameter and returns an integer can be given the type in Figure 2.

The universal type quantifies over the type of the portion of the caller's stack that the callee should not modify. This is the part below the argument. Callee save registers are ignored here, though they are handled with singleton types in current TAL versions.

If the start address of the function has this type, then we know that for any stack type substituted for α, the function will run without modifying anything below its argument, because the type of this part of the stack is abstract within the function. The register typing for this code pointer shows that a valid return pointer is on top of the stack upon entry. Its validity is expressed through the fact that it expects `eax` to hold a word, matching with the x86 convention for returning a value; and it expects the stack to be restored to its state before the call, expressed using the universally quantified stack type. Since α appears only as the tail of the function entry and function return stacks, it is guaranteed that the tail of the stack upon return must in fact be

$$\exists \Gamma, L, T, m, v_1, ..., v_n.pc = L \wedge mem = m \wedge r_1 = v_1 \wedge ... \wedge r_n = v_n \wedge \Gamma \vdash_m L : \texttt{code}\ T \wedge \Gamma \vdash_m \{r_1 = v_1, ..., r_n = v_n\} : T$$

Figure 3: The global progress continuation

the original tail. The function lacks enough typing information to construct new values of this stack type if it is to typecheck.

One important thing to note is that TAL's strategy of using general code types to describe functions allows a great deal of freedom in choosing calling conventions. The standard x86 convention can be a good choice based on instruction set support for it, but any well-typed convention is possible and enforceable by TAL typecheckers.

4. VERIFYING A FUNCTION CALL WITH THE TALX86 OPEN VERIFIER EXTENSION

The TALx86 Open Verifier extension I am developing is based on the TALx86 library provided with the TALC 1.0 release from Cornell University. These libraries perform typechecking of TAL code using custom representations and logic. The job of a TAL extension is to express the involved information in a first-order logic form and prove the soundness of the lemmas that trusted TAL typecheckers take for granted in verifying a program's type safety.

4.1 Definitions of predicates

Kind judgments translate trivially into a recursive definition of a first-order has-kind relation.

Any expression may treated as a machine word, so $\Gamma \vdash_m e : \texttt{word} \equiv \texttt{true}$.

$\texttt{stackptr}(\texttt{nil})$ may be treated similarly, since a `nil` stack will never be accessed. $\Gamma \vdash_m e : \texttt{stackptr}(\texttt{cons}(\tau_1, \tau_2))$ iff $(\texttt{addr}\ e)$ and $\Gamma \vdash_m (\texttt{sel}\ m\ e) : \tau_1$ and $\Gamma \vdash_m e + 4 : \texttt{stackptr}(\tau_2)$. Here `addr` is a built-in unary predicate on expressions that is true only for an expression that indicates the memory address of a machine word available for reading and writing. The fact that every address a program accesses satisfies this predicate is the basic definition of the safety policy the Open Verifier enforces. The built-in binary `sel` function maps from a memory state and an address to the contents of the machine word at that address.

$\Gamma \vdash_m e : \alpha$ iff $\alpha = \tau \in \Gamma$ and $\Gamma \vdash_m e : \tau$. Universal typing judgments can be given the obvious meanings in terms of universal quantification over types in first-order logic.

The only values TAL gives code type are code labels. Every program has a fixed set of code labels, where each has code type wrapped in zero or more levels of universal types, abstracting such things as caller stacks and regular types for parametric polymorphism. Thus, $\Gamma \vdash_m e : \texttt{code}\ T$ can be defined to be true iff e equals a code label L and T is the result of substituting constructors of appropriate kinds for the universally quantified type variables in the body of the universal type that is L's type annotation in the program. This typing of labels can be made part of Γ through a simple extension not given in the language grammar.

The register set typing judgment's definition follows from that of the expression typing judgment.

4.2 Lemmas

A few lemmas are critical for proving function call soundness. There are the obvious introduction and elimination rules for stack pointer types that arise directly from the corresponding predicate definitions. There is also the obvious elimination rule for universal types, based on syntactic substitution for the type variable the universal type binds. No introduction rule for universal types is provided; the current implementation only allows them to arise from program code labels. There are also lemmas that define register set type-value compatibility recursively.

None of these lemmas seem to pose any great proof burden. It seems likely that standard first-order logic techniques allow straightforward proofs of them all, although they have not yet been proven for the TALx86 extension.

4.3 The global progress continuation

A single progress continuation is needed for verifying TAL programs. It says "it's OK to jump to any value of code type as long as the registers have values of appropriate types." Formally, where $r_1, ..., r_n$ is the complete set of registers, the progress continuation is expressed by the formula in Figure 3. The occurrences of pc, mem, and $r_1, .., r_n$ in the formula are constants to which a concrete machine state gives values, corresponding to the current program counter value, the memory contents, and the values of the registers, respectively.

At the beginning of verification, this continuation is proved covered by the set of all program verification roots corresponding to TAL labels: Since the notion of having code type is defined in terms of equality to a code label, the first typing predicate in the continuation is equivalent to a disjunction over L's equality to the labels of the program, quantifying over the possible substitutions for bound type variables in each case. The register set typing lemmas allow the unrolling of the second predicate to show for each disjunct that the second typing predicate implies the known facts about the registers in the corresponding verification root's first-order state. Each such root contains predicates giving *only the kinds* of the universally bound type variables of an existentially quantified context. Thus, this context may be instantiated to give any concrete type variable substitutions consistent with this kind information. This allows proofs of coverage for each disjunct of the global continuation, meaning that every machine state satisfying the global continuation also satisfies the first-order formula associated with one of the program states that has been queued to be verified safe before declaring the whole program safe. It is clear intuitively that this implies that a concrete state satisfying the progress continuation's formula may be declared safe at any point during verification.

4.4 Coverage proofs for function call jumps

Consider a function label L with the example one-argument function type of Figure 2. Directly before a jump to the function entry point, the argument and return pointer have been pushed onto the stack. The TAL extension will make

$$\texttt{stackptr}(\texttt{cons}(\texttt{code}\ \{\texttt{eax} : \texttt{word}, \texttt{esp} : \texttt{stackptr}(\texttt{cons}(\texttt{word}, \tau))\}, \texttt{cons}(\texttt{word}, \tau)))$$

Figure 4: The type of register esp directly before a call to a function with the type from Figure 2, where τ describes the part of the stack the function will ignore

$$\texttt{code}\ \{\texttt{esp} : \texttt{stackptr}(\texttt{cons}(\texttt{code}\ \{\texttt{eax} : \texttt{word}, \texttt{esp} : \texttt{stackptr}(\texttt{cons}(\texttt{word}, \tau))\}, \texttt{cons}(\texttt{word}, \tau)))\}$$

Figure 5: A type for a function with the type from Figure 2, where the universally bound type variable has been instantiated to τ

sure it can prove $\Gamma \vdash_m sp : \sigma$, where sp is the current value of esp, τ a type known to describe the appropriate part of the stack, and σ the type in Figure 4. The type of the return pointer label is part of Γ, and its inclusion in Γ is justified by including every possible return pointer in the program as a verification root.

The universal type elimination rule can be used to show that, since L has the type from Figure 2, it has the type in Figure 5. The register set typing lemmas allow a straightforward proof that the current register values are compatible with this code type's register typing. Thus, using the clear instantiations of the global continuation's existential variables, we have the needed coverage proof showing that the entry state for the function call satisfies the global continuation formula and is thus safe.

The critical aspect of this is that *the existential variable instantiations are not being chosen for the called function's verification root*. They are being chosen for the meta-root represented by the global continuation. This allows the current Γ to be used as the new instantiation, despite the fact that different functions will generally need different typing contexts. It is the coverage proof for the global continuation at the beginning that does all the work of proving this sound.

4.5 Other coverage proofs

Function returns and normal jumps use the same basic method as above. All code labels may be treated in the same way for these purposes. Functions bodies are just code blocks that happen to use continuation passing style. Function returns are simply jumps to first-class continuations that have been saved on the stack.

5. CONCLUSIONS

One of the central open questions about the Open Verifier architecture is how well progress continuations can be used to encode different models of program state. If the Open Verifier is to meet its goals as a flexible verification platform, progress continuations should be flexible enough to encode any higher-order properties relevant to the safety of particular languages. My experiences to date with the TAL extension seem to indicate that the Open Verifier can indeed handle nontrivial uses of higher-order program properties, including first-class functions and parametric polymorphism.

The current implementation requires that a number of key components be trusted. Calls to runtime system functions are assumed safe, proofs that the initial verification roots chosen by an extension cover the real program entry point are omitted, and formal proofs have not yet been constructed for the lemmas used at each verification step. The process of proving safety of interactions with a complete runtime system that includes a realistic garbage collector remains a research problem. The extension modifications needed to add the other two missing aspects seem to be straightforward.

With these omissions, the current TAL extension handles programs compiled from Popcorn (a safe C dialect) and Scheme, using the compilers included in the TALC 1.0 release. The interesting types and constructs are handled in a reasonably straightforward way that is close to the way a standard typechecker would be implemented. These results provide good evidence that the Open Verifier supports construction of diverse kinds of extensions without requiring proofs involving indexed relations or other unintuitive techniques.

6. REFERENCES

[1] The CLI architecture. Technical Report ECMA TC39/TG3, ECMA, October 2001.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.

[3] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Programming Languages and Systems*, 23(5):657–683, 2001.

[4] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[5] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. 1999.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[7] George 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, January 1997.

[8] George C. Necula and Robert Schneck. Proof-carrying code with untrusted proof rules. In *Proceedings of the 2nd International Software Security Symposium*, November 2002.

[9] George C. Necula and Robert Schneck. A sound framework for untrustred verification-condition generators. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS03)*, July 2003.