The Open Verifier Framework for Foundational Verifiers*

Bor-Yuh Evan Chang[1]
bec@cs.berkeley.edu

Adam Chlipala[1]
adamc@cs.berkeley.edu

George C. Necula[1]
necula@cs.berkeley.edu

Robert R. Schneck[2]
schneck@math.berkeley.edu

[1]Department of Electrical Engineering and Computer Science
[2]Group in Logic and the Methodology of Science
University of California, Berkeley

ABSTRACT

We present the Open Verifier approach for verifying untrusted code using customized verifiers. This approach can be viewed as an instance of foundational proof-carrying code where an untrusted program can be checked using the verifier most natural for it instead of using a single generic type system. In this paper we focus on a specialized architecture designed to reduce the burden of expressing both type-based and Hoare-style verifiers.

A new verifier is created by providing an untrusted executable extension module, which can incorporate directly pre-existing non-foundational verifiers based on dataflow analysis or type checking. The extensions control virtually all aspects of the verification by carrying on a dialogue with the Open Verifier using a language designed both to correspond closely to common verification actions and to carry simple adequacy proofs for those actions.

We describe the design of the trusted core of the Open Verifier, along with our experience implementing proof-carrying code, typed assembly language, and dataflow or abstract interpretation based verifiers in this unified setting.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Security

Keywords

Proof-Carrying Code, Typed Assembly Language, Language-Based Security

*This research was supported in part by NSF Grants CCR-0326577, CCR-0081588, CCR-0085949, CCR-00225610, and CCR-0234689; NASA Grant NNA04CI57A; a Sloan Fellowship; an NSF Graduate Fellowship; an NDSEG Fellowship; and California Microelectronics Fellowships. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

TLDI'05, January 10, 2005, Long Beach, California, USA.

1. INTRODUCTION

Since its introduction [AF00], foundational proof-carrying code (FPCC) has become a standard. In FPCC, a code consumer need trust only a proof checker for some low-level logic, as well as axioms in that logic defining the operational semantics of a particular architecture and the desired notion of safety. Untrusted code comes with a proof directly expressing the desired result: that the code (expressed as a sequence of machine words) satisfies the predicate defining safety. This proof is called *foundational* in that it must prove safety directly in terms of the machine semantics; in particular, such notions as type safety, rather than being trusted, must be defined by the code producer and proven to imply the low-level safety policy.

FPCC is remarkable in its simplicity and in its generality; it is completely flexible with respect to what safety policy is defined and how a given program is shown to meet that safety policy. However, this observation comes with a big "in principle", for FPCC is only useful if the foundational proofs can be feasibly constructed for a broad class of programs. The research in FPCC [AF00, HST$^+$02, Cra03] has naturally focused on this very question. Thus far there has been relatively little advantage taken of the generality of FPCC, with most papers discussing how to prove memory safety using some foundationalization of typed assembly language (TAL). Different groups have put forth different logical perspectives on how to do this. Nonetheless, it remains the case that after a number of years of FPCC research, very few systems have been constructed.

The following questions and observations motivate our research.

- Can we construct a simple yet reasonably complete framework that significantly eases the engineering effort required to create a new FPCC instantiation? In particular, rather than producing a single FPCC backend to be used for as many source languages as possible, can we make it feasible to develop for each new language a new FPCC instantiation that is more naturally suited to it?

- Past efforts have considered development of new certifying compilers that produce foundational proofs for their output. How would techniques differ if we set the goal of writing verifiers for existing compilers over which we have little control?

1. $I_0 \implies \bigvee_{i=1,\ldots,n} I_i$, and
2. for each $i = 1, \ldots, n$
   (a) $I_i \implies \mathit{SafeState}$, and
   (b) $\mathit{Post}(I_i) \implies \bigvee_{j=1,\ldots,n} I_j$

Figure 1: Generic code verification strategy

- The language of past FPCC work has been aimed at programming language researchers. One can expect that it is beyond the understanding of most engineers who might be called upon to implement foundational verifiers in an industrial setting. To what extent can we make the ideas of FPCC more widely accessible? In particular, how much can FPCC proofs remain natural yet avoid the use of heavy-duty logical tools?
- The basic FPCC checking process runs in two stages: a proof generator builds a single proof for a program's safety, and then a proof checker verifies that it proves what it should. Can we design a feasible alternate interaction that helps remove concerns about transferring large proofs over a network, and reduces other overhead required to check monolithic proofs?

In this paper, we present the *Open Verifier*, a system that we believe partially answers these questions. We have two complementary perspectives on the Open Verifier. First, it can be viewed as a new proof-construction strategy for pure FPCC in the usual sense. Second, it can be viewed as a novel architecture, in principle more restricted than FPCC though practically sufficient, which exhibits specific engineering benefits.

On the one hand, we consider the Open Verifier to be a new logical perspective on how to construct foundational proofs for safety properties such as memory safety. The superstructure of the proofs is simple: starting with some state predicate describing the initial states of a program, we symbolically execute the program by iteratively producing the strongest postcondition (relative to the safety policy). At each stage, however, the particular enforcement mechanism (designed by the code producer) determines some provably *weaker* predicate to replace the strongest postcondition.

More formally, we must prove that the execution of the machine proceeds safely forever when started in a state that satisfies the initial predicate I_0. The safety policy is described by a predicate on states *SafeState*, and the semantics of one execution step is modeled as a predicate transformer function *Post*, such that $\mathit{Post}(I)$ is the predicate describing the state of the machine after making one step from a state described by I. The actual verification relies on a finite set of invariant predicates I_i ($i = 1, \ldots, n$), typically one per untrusted instruction, along with the proofs of the facts shown in Figure 1. Once we have these proofs, it is a relatively easy task to prove, by induction on the number of execution steps, that the execution proceeds safely forever to states that satisfy both *SafeState* and the invariant $\bigvee_{i=1,\ldots,n} I_i$.

This approach exhibits the common structure of a variety of code verification strategies, ranging from bytecode verification, as in today's JVML or CIL machines, to typed-assembly language, or to proof-carrying code, including foundational proof-carrying code. The Open Verifier architecture can be customized to behave like any of these techniques.

From the pure FPCC perspective, the next question is how to organize the weakening of the strongest postcondition at each step. Here we emphasize customizability; the approach may depend on anything from Hoare logic, to abstract interpretation or dataflow analysis based on complex type systems. When we do use type systems, we advocate a syntactic and intensional approach roughly similar to that of [HST+02], but rather than using a global well-typedness notion over abstract states and a bisimulation argument to connect to concrete states, we define each type directly as a predicate over concrete values. Furthermore, rather than massaging each source language into some single type system using special certifying compilers, we intend that the verifier for each language be written in the most natural way, each with its own type system; in fact, we advocate the re-use of existing compilers and verifiers as much as possible, having found it often possible to modularly add a foundationalizing layer. All of these features, we believe, combine to make the logical perspective of the Open Verifier project a very accessible approach to proof construction for FPCC.

On the other hand, we also bring a particular engineering perspective to this project. Below we mention three aspects of this. These engineering considerations require certain restrictions on which safety policies can be considered and how proofs can be constructed. It might be felt that these restrictions are a step back from the generality of FPCC. This is true, in principle; and in principle our logical perspective can be used independently of our engineering concerns. In practice, however, we believe that all existing approaches to constructing FPCC proofs already work within these restrictions.

First, we suggest a different mode of interaction where, instead of shipping the complete foundational proof, a code producer can send an untrusted, executable *verifier extension*, specific to a given source language and compilation strategy, which works with trusted parts of the Open Verifier architecture to produce necessary parts of the foundational proof in an on-line manner. In essence, the extension incorporates the proof-generation schemas common to a source language. This approach obviates typical FPCC concerns about proof representation and proof size [NR01], while allowing extensions to choose how much proof and in what form is actually carried by the code. The work of [WAS03] also suggests that the untrusted agent provide an executable verifier, in the form of a logic program with an accompanying soundness proof; we accept arbitrary executable verifiers, which must be provably safe to execute. While our approach requires separate soundness proofs for individual runs, it also allows the practical use of a wider variety of verification strategies relative to particular verification-time resource constraints.

Second, we do not require that features common to virtually all verifications be completely foundational. In particular, we are willing to work with assembly code instead of machine code; we are also willing to work with a strongest postcondition generator expressed as executable code instead of logical axioms describing machine transitions. It should be obvious that foundationalizing these aspects would be straightforward using such work as [MA00], but these choices have allowed us to focus on the non-boring parts of writing a foundational verifier.

The third aspect of our engineering perspective is the most important. The most common form of "proof-carrying code" is actually bytecode verification as used in the Java Virtual Machine Language (JVML) [LY97] or the Microsoft Common Intermediate Language (CIL) [GS01, Gou02]. Writing a bytecode verifier is relatively simple in terms of number of Ph.D.s required; even working at the assembly code level, writing a verifier using type-based dataflow analysis is relatively accessible. We wish to minimize the cost of adapting even these most accessible verifiers to a foundational framework. To this end we have created a scripting language that can be used to simultaneously create each necessary new weakened invariant, together with the foundational proof that it is weaker than the strongest postcondition. The operations of the scripting language are inspired by the common operations of a type-based dataflow verifier; in our experience we have found it to be a natural technique for building foundational verifiers.[1]

We have had substantial success in producing foundational verifiers in the Open Verifier framework. We have a complete foundational verifier for TALx86 programs, as produced by existing compilers for Popcorn (a safe C dialect) and mini-Scheme; it makes direct use of the existing TALx86 verifier.[2] We have another foundational verifier for Cool, a mini-Java, relative to a trusted run-time system.

Because of space limitations, we cannot fully discuss every feature of the Open Verifier approach or fully discuss our experimental results here; an expanded version is under preparation. Here we concentrate on the structure of the Open Verifier architecture, described in Section 2. Then, in Section 3, we describe our experience implementing proof-carrying code, typed assembly language, and dataflow verifiers in this unified setting. We conclude in Section 4 with a discussion of implementation and results.

2. THE OPEN VERIFIER

The main design characteristics of the Open Verifier are as follows:

- A new verifier can be created even by untrusted users by installing into the Open Verifier an executable module, called an *extension*. There is no restriction on the algorithms or data structures that the extension can use internally to perform the verification.
- The extension can control the verification to a very large degree by answering queries from the Open Verifier. In order to ensure soundness, the extension must present a proof along with each answer. The query and answer language is designed to simplify the development of verifiers, to the point where many extensions can be simple wrappers for existing dataflow or type-based verifiers.
- In order to simplify the proof construction effort, the Open Verifier manipulates formulas and proofs of Horn logic for each verification step, using lemmas that are proved once using a more powerful logic.

[1] By making the scripting language part of the trusted framework, we also manage to reap engineering benefits in terms of verification time; for example the scripts are organized so that proving $A \wedge B \wedge C$ from $A \wedge B$ requires only a proof of C, without incorporating the trivial re-proving of A and B required in a straightforward logical formalism.

[2] We convert TALx86 to actual x86 assembly code using a simple non-garbage-collected memory allocator; foundationalizing a more realistic run-time system is future work.

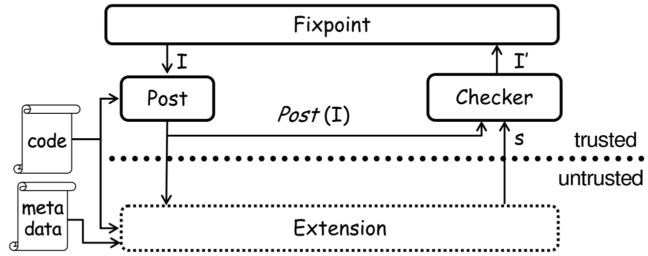

Figure 2: The Open Verifier architecture.

- Multiple extensions can coexist and can even cooperate for the verification of a program.

As observed in the Introduction, this system could with minor modification be viewed simply as a proof-construction mechanism for a pure FPCC framework; here we discuss its potential as an alternate framework to emphasize possible engineering benefits.

Figure 2 shows the interaction between various components of the Open Verifier. The modules above the dotted line are part of the trusted infrastructure. The verification process is concerned with the construction of a set of invariants I_i, with the properties shown in Figure 1. The trusted Fixpoint module collects these invariants, starting with the built-in initial invariant I_0. For each collected invariant I, we compute a trusted strongest postcondition $Post(I)$. For most verification purposes, $Post(I)$ contains too much information. For example, it may say that the value of a register is equal to the result of adding two other registers, while for verification purposes all that is needed is that the result has integer type. The purpose of the untrusted Extension module is to provide one or more weaker invariants, along with proofs that they are indeed weaker. One of the contributions of this work is to identify a scripting language in which the extension module can describe the incremental weakening changes that must be made to the strongest postcondition in order to produce the desired invariants. The basic scripting operations correspond closely with the actions of a type-based verifier. A script could require abstracting the value of a register, dropping a known fact, or even adding new assumptions. In the last case, the script must also specify a proof of the newly added assumptions. Thus, the invariants generated are weaker than the strongest postcondition by construction. The trusted Checker module interprets the scripts and checks the accompanying proofs.

The extension incorporates the safety-mechanism-specific reasoning that is necessary for verification, and may take advantage of metadata that accompanies the code. This metadata acts as a private channel between the producer of the untrusted code and the extension. The metadata can range from entire proofs demonstrating the safety of particular instructions, to simply the types of the inputs and outputs of all the functions in the program. The high-level extensions we have implemented are all similar to this second case, where the cost of producing and transmitting the metadata is negligible.

2.1 Verification Example

Here we present a simple example to which we will refer throughout. Suppose we are working with lists of integers, represented as either 0 for an empty list, or the address of a

```
1  rc := 0
2  if rs = 0 jump 7
3  rh := load[rs]
4  rs := load[rs + 1]
5  rc := call cons rh rc
6  jump 2
7  return
```

Figure 3: A function taking a list $\mathbf{r}_\mathrm{s}$ and producing the reverse list $\mathbf{r}_\mathrm{c}$.

two-word cell in memory containing, respectively, an integer head and a list tail. Call this type system L; we might use C or a specialized source language to work with L. We assume the existence of an allocation function `cons` taking an integer and a list and producing the required two-word cell.

The assembly-language code shown in Figure 3 has the effect of taking the list $\mathbf{r}_\mathrm{s}$ and producing the reversed list $\mathbf{r}_\mathrm{c}$. Although we are able to verify uses of stack frames, for this presentation we assume that the function argument $\mathbf{r}_\mathrm{s}$ and return value $\mathbf{r}_\mathrm{c}$ are implemented as registers that are preserved by `cons`. We similarly ignore for this presentation the issue of manipulating the return address register and otherwise setting up function calls. These issues are addressed (by discussing their handling in a particular fully developed extension) briefly in Section 3.2, and in substantial detail in [Chl04].

In the following section, we will indicate how to produce an extension for L.

2.2 The Invariants

As described in the Introduction, verification proceeds by creating invariants—predicates on machine states—for various points during program execution. The invariants have a special syntactic form shown in Figure 4.

The invariants are triples $(\Gamma; R; \Psi)$, where R specifies symbolic expressions for all machine registers, Ψ specifies a number of named formulas, and Γ binds the typed existential variables that can be used in R and Ψ.

Formally, an invariant $(\Gamma; R; \Psi)$ denotes the following predicate on the machine registers:

$$\exists_{x_i:\tau_i\in\Gamma} x_i : \tau_i \, . \left(\bigwedge_{r_j=e_j\in R} r_j = e_j \;\wedge \bigwedge_{h_k:\phi_k\in\Psi} \phi_k \right)$$

The effect of this syntactic form is just that register names cannot be involved directly in assumptions, but only indirectly by means of existential variables in terms of which they are defined.

We assume that the machine has a finite set of registers, and a special "memory" register $\mathbf{r}_\mathrm{m}$ that describes the contents of the memory. The expression $(\mathsf{sel}\ m\ a)$ denotes the contents of memory denoted by m at address a.

The Open Verifier defines the type mem used for memory expressions and the contents of memory pseudo-registers, and val for the other expressions. We also assume that there are a number of extension-defined base types b; in our implementation, we allow the use of inductively defined types. The expression constructors f and formula constructors a may take only expressions as arguments. A number of these constructors are provided by the Open Verifier (e.g., the $\mathsf{sel} : \mathsf{mem} \to \mathsf{val} \to \mathsf{val}$ expression constructor, and the $= : \mathsf{val} \to \mathsf{val} \to \mathsf{Prop}$ formula constructor). In our experiments, the Open Verifier enforces memory safety by defining a formula constructor Addr that holds on valid addresses.

$$\begin{array}{rlll}
\text{invariants} & I & ::= & (\Gamma; R; \Psi) \\
\text{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\
\text{register files} & R & ::= & \cdot \mid R, r = e \\
\text{assumptions} & \Psi & ::= & \cdot \mid \Psi, h : \phi \\
\text{variables} & x & & \\
\text{hypotheses} & h & & \\
\text{registers} & r & ::= & \mathbf{r}_1 \mid \mathbf{r}_2 \mid \ldots \mid \mathbf{r}_\mathrm{m} \\
\text{expressions} & e & ::= & x \mid n \mid \ f\ e_1 \ldots e_n \mid \ldots \\
\text{base types} & b & & \\
\text{types} & \tau & ::= & \mathsf{val} \mid \mathsf{mem} \mid b \\
\text{formulas} & \phi & ::= & \top \mid \bot \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \\
 & & & \mid \ a\ e_1 \ldots e_n \mid \ldots \\
\text{constructors} & n & : & \mathsf{val} \\
 & f & : & \tau_1 \to \cdots \to \tau \\
 & a & : & \tau_1 \to \cdots \to \mathsf{Prop}
\end{array}$$

Figure 4: Syntactic entities used by the framework.

Any logic could be used for the formulas; in practice we have found it sufficient and convenient to work with a restricted set of propositional connectives, together with formula constructors defined by the extension in a richer logic (see Section 2.7 for further discussion).

Now we discuss invariants for L. The L extension defines typing relative to an allocation state of type lalloc, whose members are sets of addresses that have been allocated to hold lists. The L extension will define the following formula constructors:

$$\begin{array}{c}
\mathsf{LInv} : \mathsf{lalloc} \to \mathsf{mem} \to \mathsf{Prop} \\
\mathsf{List} : \mathsf{lalloc} \to \mathsf{val} \to \mathsf{Prop} \\
\mathsf{Nelist} : \mathsf{lalloc} \to \mathsf{val} \to \mathsf{Prop}
\end{array}$$

We write $(\mathsf{LInv}\ a_L\ m_L)$ to say that the contents of the L-memory state m_L are consistent with the allocation state a_L, and we write $(\mathsf{List}\ a_L\ \ell)$ and $(\mathsf{Nelist}\ a_L\ \ell)$ to say that ℓ is of type list, or non-empty list respectively, in allocation state a_L. The extension defines these predicates as follows:

$$\begin{array}{l}
\mathsf{LInv}\ a_L\ m_L \iff \\
\qquad \forall \ell \in a_L.\ (\mathsf{Addr}\ \ell) \wedge (\mathsf{Addr}\ (\ell + 1)) \wedge \\
\qquad\qquad \big(\mathsf{List}\ a_L\ (\mathsf{sel}\ m_L\ (\ell + 1))\big)
\end{array}$$

$$\begin{array}{r}
\mathsf{Nelist}\ a_L\ \ell \iff \ \ell \in a_L \\
\mathsf{List}\ a_L\ \ell \iff \ \ell = 0 \vee \ell \in a_L
\end{array}$$

We note that the typing predicate is defined *intensionally* by means of a typing context (the allocation state), rather than *extensionally* by direct reference to the memory; thus we are closer to the "syntactic" approach of [HST+02], as opposed to the "semantic" approach of [AF00]. We believe that this approach makes the development of extensions more accessible.

With these definitions the invariant for the instruction 1 in Figure 3 may be written as[3]

$$\begin{array}{l}
\Gamma = x_1 : \mathsf{val}, \ldots, x_\mathrm{m} : \mathsf{mem}, x_\mathrm{a} : \mathsf{lalloc}; \\
R = \mathbf{r}_\mathrm{pc} = 1, \mathbf{r}_1 = x_1, \ldots, \mathbf{r}_\mathrm{s} = x_s, \\
\qquad \mathbf{r}_\mathrm{m} = x_\mathrm{m}; \\
\Psi = h_L : \mathsf{LInv}\ x_\mathrm{a}\ x_\mathrm{m},\ h_s : \mathsf{List}\ x_\mathrm{a}\ x_s
\end{array}$$

[3]This is not the initial invariant I_0 from Figure 1, which would be a primitive description of the initial machine state, independent of any particular extension. In practice the extension will have to prove that this invariant is weaker than the actual I_0.

When $P(R(\mathbf{r}_{\rm pc}))$ is "$\mathbf{r}_1 := n$":
$\{(\Gamma;\ R, \mathbf{r}_{\rm pc} = R(\mathbf{r}_{\rm pc}) + 1, \mathbf{r}_1 = n;\ \Psi)$

When $P(R(\mathbf{r}_{\rm pc}))$ is "$\mathbf{r}_1 := \mathbf{r}_2$":
$\{(\Gamma;\ R, \mathbf{r}_{\rm pc} = R(\mathbf{r}_{\rm pc}) + 1, \mathbf{r}_1 = R(\mathbf{r}_2);\ \Psi)\}$

When $P(R(\mathbf{r}_{\rm pc}))$ is "$\mathbf{r}_1 := \mathtt{load}[\mathbf{r}_2]$":
$\{(\Gamma;\ R, \mathbf{r}_{\rm pc} = R(\mathbf{r}_{\rm pc}) + 1, \mathbf{r}_1 = (\mathsf{sel}\ R(\mathbf{r}_{\rm m})\ R(\mathbf{r}_2));$
$\Psi, h_{safe} : (\mathsf{Addr}\ R(\mathbf{r}_2))),$
$(\Gamma;\ R, \mathbf{r}_{\rm pc} = \mathsf{error};\ \Psi, h_{unsafe} : \neg(\mathsf{Addr}\ R(\mathbf{r}_2)))\}$

When $P(R(\mathbf{r}_{\rm pc}))$ is "$\mathtt{if}\ \mathbf{r}_1 = 0\ \mathtt{jump}\ n$":
$\{(\Gamma;\ R, \mathbf{r}_{\rm pc} = R(\mathbf{r}_{\rm pc}) + 1;\ \Psi, h_{branch} : (R(\mathbf{r}_1) \neq 0)),$
$(\Gamma;\ R, \mathbf{r}_{\rm pc} = n;\ \Psi, h_{branch} : (R(\mathbf{r}_1) = 0)\}$

Figure 5: Examples of Post(Γ; R; Ψ) for memory safety.

In the remaining description of the example, we omit the allocation state argument to the above constructors—it does in fact change at the call to `cons`, but not at any of the transitions we will work out in detail here. Also, whenever an existential variable is the value of a register, we use that register name instead of the variable in symbolic expressions and formulas. Since the various components of an invariant are syntactically distinct, we will freely mix in the examples below elements of Γ, R, and Ψ. With these simplifications, the above invariant can be written simply as

$$I_1 \ = \ \mathbf{r}_{\rm pc} = 1, h_L : \mathsf{LInv}\ \mathbf{r}_{\rm m},\ h_s : \mathsf{List}\ \mathbf{r}_{\rm s}.$$

2.3 The Trusted Post Module

In order to be able to verify the claims of the untrusted extensions, the Open Verifier uses a trusted strongest-postcondition generator, the Post module, to interpret instructions. Understood at a high level, Post takes a state predicate and produces the strongest postcondition, a new state predicate that is guaranteed to hold of the successor state of any state satisfying the input.

To model the safety policy, we introduce a special value error for the program counter and assume every unsafe transition sets the program counter to error. A state is safe (meeting the predicate *SafeState* of Figure 1) if its program counter is not error; a program is safe if every state reachable from an initial state of the program is safe.[4] This formulation makes it easy to enforce safety using Post. A potentially unsafe transition is treated like a branch statement, with a disjunctive postcondition; either the transition is safe and execution continues as expected, or the transition is unsafe and the execution proceeds to a special error state.

For Post to be usable in our framework, it needs to work with the syntactic form of invariants. This is done by interpreting a disjunctive postcondition as *two* output invariants. All the other standard strongest postconditions already preserve the form of our invariants; this is part of the motivation for using this form. Finally, we augment the safety policy to require that the code of the program being verified may not be overwritten. This simplification, which of course holds for all realistic examples, allows us to have the trusted Post module determine the instruction to be executed from the program counter.

Fix a program P to be verified. Given an invariant I, the Post module reads the value of the program counter register from I, reads the corresponding instruction from P, and produces one or two successors invariants $\bar{I}$. Most instructions yield one successor invariant. The exceptions are the branch instructions and those instructions whose usage is restricted by the safety policy, as described above. In our example, we assume that only memory operations are restricted, requiring that the predicate Addr holds for the address being accessed.

Figure 5 shows examples from the definition of the Post function for memory safety. These are simply the standard definitions for a strongest postcondition expressed as Open Verifier invariants. We use the notation $R(\mathbf{r}_{\rm i})$ to denote the value associated with $\mathbf{r}_{\rm i}$ in R, and we write $R, \mathbf{r}_{\rm i} = e$ to denote a modified register file that is like R except $\mathbf{r}_{\rm i}$'s value is e. Similarly, we write $\Psi, h : \phi$ to denote a set of assumptions in which the assumption named h is replaced with ϕ. The post-states of the branch instruction contain assumptions about the outcome of the guard predicate.

The Post module may be invoked only on invariants that assign to the program counter register a literal value n that denotes an address in the instruction stream. In all other cases, Post aborts the verification. This means that extensions will have to show that the error successor invariants are not reachable. For the load case ($\mathbf{r}_1 := \mathtt{load}[\mathbf{r}_2]$), the extension ought to produce a proof that ($\mathsf{Addr}\ R(\mathbf{r}_2)$), which along with the h_{unsafe} assumption can be used to derive a contradiction and hence prove that the post-state is not really reachable. Since indirect jumps result in postconditions where the program counter is not a constant, the extension must essentially weaken these to the disjunction of all possible literal program counters; this is discussed further in Section 3.2.

2.4 The Untrusted Extensions

The Post module produces a set of invariants that together cover all the possible successor states without losing any information. In order for the verification to terminate, we must occasionally abstract. Abstraction is also useful to simplify or to speed up the verification process. In the Open Verifier, abstraction is fully customizable and is performed by the untrusted extensions.

For each invariant I' returned by the Post module, the extension must do one of the following things: (a) prove that I' is contradictory and thus it does not describe any concrete state, (b) prove that a weaker invariant I_i is already part of the collection of invariants computed by Fixpoint, (c) tell the Fixpoint module to add a weakened version of I' to the collection, or (d) perform a combination of these actions using a case analysis on the invariant I'.

The Fixpoint module starts with the initial invariant I_0, and grows a collection Δ of named invariants,[5] using the following syntax:

$$\begin{array}{lcl} \text{names} & u & \\ \text{collections} & \Delta & ::= \ \emptyset \mid \Delta, u : I \end{array}$$

Intuitively, the extension must yield, for each $I' \in \mathrm{Post}(I)$, a new collection Δ', along with a proof that the disjunction of the invariants in Δ' (together with any previously collected Δ) is weaker than I', and thus it is satisfied in the

[4]This is clearly equivalent to an alternate formulation of safety whereby unsafe transitions are made impossible, and safety is defined as making progress forever.

[5]Invariants in the collection are always referred to by name. When a new invariant is added, the Fixpoint module does not check equivalence with old invariants. It is the extension's job to ensure that the verification terminates by proving equivalence with an old invariant rather than adding a new invariant.

successor state. We make two observations. First, we ought to describe the invariants in Δ' by incremental changes from I'. This simplifies our work; it also allows an extension to carry along any components of I' that it does not understand because they were introduced by other cooperating extensions, which is a subject of ongoing research for us. Second, the weakening proofs that are required are typically large structures whose leaves are interesting applications of extension-specific lemmas, while the rest is a tedious construction whose sole purpose is to match the particular structure of I' and Δ'.

Based on these observations, we have designed a language of *scripts* that extensions can use to describe incremental changes to the post-invariant. Each change corresponds to a class of common verification actions, and some carry proofs that justify these actions. The scripting language is shown below. Though we present it here as a programming language, in our implementation, extensions indicate the "scripts" they want executed by calling one API function per construct of the language below.

$$\begin{array}{rrcl}
\text{scripts} & s & ::= & \textbf{abstract } e \textbf{ as } x \textbf{ in } s \\
&&& \mid \textbf{set } r = e \textbf{ by } \gamma \textbf{ in } s \\
&&& \mid \textbf{assert } h : \phi \textbf{ by } \gamma \textbf{ in } s \\
&&& \mid \textbf{forget } h \textbf{ in } s \\
&&& \mid \textbf{collect as } u \\
&&& \mid \textbf{match } u \\
&&& \mid \textbf{unreachable by } \gamma \\
&&& \mid \textbf{cases } \gamma \textbf{ of } h_1 \Rightarrow s_1 \mid h_2 \Rightarrow s_2 \\
\text{proofs} & \gamma & ::= & h \mid g\, e_1 \ldots e_n\, \gamma_1 \ldots\, \gamma_m \mid \ldots \\
\text{rules} & g & : & \Pi x_1{:}\tau_1. \ldots \Pi x_n{:}\tau_n.\, \phi_1 \to \cdots \to \phi_m \to \phi
\end{array}$$

The first four script operations are atomic weakening actions: abstracting the value of an expression by introducing a fresh existential variable, changing the value of a register along with a proof that the new value is an abstraction of the old one, adding a new assumption along with its proof, and forgetting an assumption. Proofs are constructed from names of assumptions using proof rules. The Open Verifier provides a few trusted proof rules, while most of the proof rules are introduced, and justified, by the extension. We use the judgments-as-types principle [ML71, HHP93, CH85] to describe proof rules using a dependent-type notation in order to reduce proof checking to type checking.

The remaining four script operations correspond to the main actions described above: collecting and naming an invariant, identifying the invariant with one already in the collection, showing that the invariant is unreachable, or a case analysis performing a combination of such actions. We describe next the use of scripts for our running example, and then we give the formal semantics of scripts as implemented by the trusted Checker module.

2.4.1 Extension Script and Proof Examples

We will assume that we need only verify that the function of Figure 3 is well-typed, rather than having to verify that it correctly implements a list reversal. Consider the processing of the instruction at line 1 in Figure 3. We show below the invariant from the collection that points to the instruction (I_1, derived in Section 2.2), the invariant I' after Post, and the script s' returned by the extension for handling I':

$$\begin{array}{rcl}
I_1 & = & \mathbf{r}_{\mathrm{pc}} = 1, h_L : \mathsf{LInv}\ \mathbf{r}_{\mathrm{m}},\ h_s : \mathsf{List}\ \mathbf{r}_{\mathrm{s}} \\
I' & = & I_1, \mathbf{r}_{\mathrm{pc}} = 2, \mathbf{r}_{\mathrm{c}} = 0 \\
s' & = & \textbf{abstract } 0 \textbf{ as } c \textbf{ in} \\
&& \textbf{set } \mathbf{r}_{\mathrm{c}} = c \textbf{ by } (\mathsf{ID}\ 0) \textbf{ in} \\
&& \textbf{assert } h_c : \mathsf{List}\ c \textbf{ by } \mathsf{LISTNIL} \textbf{ in} \\
&& \textbf{collect as } I_2
\end{array}$$

We will use the notation I_i to refer to either an invariant or a unique name for it, where context makes it clear which meaning applies; we use the notation $I, r = e$ (or $I, h : \phi$) to denote the result of replacing a register (or assumption) in the invariant I. Note how the post-invariant carries precise value information about $\mathbf{r}_{\mathrm{c}}$. The L extension only needs to know that the content of $\mathbf{r}_{\mathrm{c}}$ is a List; this process of "forgetting" is very typical for extensions and is essential in general to terminating verifications. So, the first operation in the script is to abstract the value of the register $\mathbf{r}_{\mathrm{c}}$ by introducing a new existential variable c into the context to stand for 0 in the expressions and formulas that are added to the invariant. Next the script sets the value of $\mathbf{r}_{\mathrm{c}}$ to be c. Each **set** operation must be accompanied by a proof that the current value of the register is equal to the new value, after accounting for the instantiation of the newly created existential variables. In this case, we need a proof that 0 (the value of $\mathbf{r}_{\mathrm{c}}$ in I') is equal to 0 (the value being set for $\mathbf{r}_{\mathrm{c}}$, after substituting c by the expression 0 that it abstracts). To construct this proof, we use the identity proof rule that is provided by the Open Verifier:

$$\mathsf{ID} : \Pi x{:}\mathsf{val}.(x = x)$$

The Checker module (described below) checks that the proof ($\mathsf{ID}\ 0$) is indeed a proof of $(0 = 0)$. The next step in the script is to add an assumption that the value of $\mathbf{r}_{\mathrm{c}}$ is a list. For this purpose, the extension defined a proof rule:

$$\mathsf{LISTNIL} : (\mathsf{List}\ 0)$$

Whenever the extension wishes to use a new proof rule, it must prove that it is derivable using the existing rules and the definitions of the formula and expression constructors that it uses. This is clearly the case for LISTNIL given the definition of List.

Using the above script the Checker module will generate the weakened version of I', say I_2, as follows:

$$I_2 \;=\; \mathbf{r}_{\mathrm{pc}} = 2, h_L : \mathsf{LInv}\ \mathbf{r}_{\mathrm{m}},\ h_s : \mathsf{List}\ \mathbf{r}_{\mathrm{s}},\ h_c : \mathsf{List}\ \mathbf{r}_{\mathrm{c}}$$

The final step in the script is to instruct the Fixpoint module to collect the weakened invariant under the name I_2.

Consider next the branch instruction at address 2 in Figure 3, where I_2 is the invariant that points to the instruction, I' and I'' are the post-invariants for the taken and fall-through branches, and s' and s'' are the corresponding scripts returned by the extension.

$$\begin{array}{rcl}
I' & = & I_2, \mathbf{r}_{\mathrm{pc}} = 8, h_{branch} : (\mathbf{r}_{\mathrm{s}} = 0) \\
s' & = & \textbf{forget } h_{branch} \textbf{ in} \\
&& \textbf{collect as } I_8 \\
I'' & = & I_2, \mathbf{r}_{\mathrm{pc}} = 3, h_{branch} : (\mathbf{r}_{\mathrm{s}} \neq 0) \\
s'' & = & \textbf{assert } h_s : \mathsf{Nelist}\ \mathbf{r}_{\mathrm{s}} \textbf{ by } (\mathsf{NELIST}\ h_s\ h_{branch}) \textbf{ in} \\
&& \textbf{forget } h_{branch} \textbf{ in} \\
&& \textbf{collect as } I_3
\end{array}$$

In the taken branch, the extension chooses to do nothing, except to forget the assumption added by Post. This is because the extension knows that its typing rules do not make use of the assumption that a list is empty. In the fall-through branch, however, the extension knows that register $\mathbf{r}_\mathrm{s}$ is a list and refines its type to a non-empty list, using the following rule:

$$\mathsf{NELIST} : \Pi\ell{:}\mathsf{val}.\ (\mathsf{List}\ \ell) \rightarrow (\ell \neq 0) \rightarrow (\mathsf{Nelist}\ \ell)$$

It is easy to see how the extension will be able to prove this lemma, using the definitions of List and Nelist from Section 2.2. The Checker module will check that the proof ($\mathsf{NELIST}\ h_s\ h_{branch}$) indeed has type ($\mathsf{Nelist}\ \mathbf{r}_\mathrm{s}$) with the assumptions present in I''.[6] Since the extension knows it will not need the h_{branch} assumption anymore, it forgets it.

Consider now the handling of the memory operation from line 4 in Figure 3. (The load from line 3 is similar but simpler.) We show below the invariant I_4 before this instruction, the post-invariants I' (for the case when the read is disallowed) and I'' (for the normal execution case), along with the corresponding scripts s' and s''.

$$\begin{array}{rcl}
I_4 & = & \mathbf{r}_\mathrm{pc} = 4, h_L : \mathsf{LInv}\ \mathbf{r}_\mathrm{m},\ h_s : \mathsf{Nelist}\ \mathbf{r}_\mathrm{s},\ h_c : \mathsf{List}\ \mathbf{r}_\mathrm{c} \\
I' & = & I_4, \mathbf{r}_\mathrm{pc} = \mathsf{error}, h_{unsafe} : \neg(\mathsf{Addr}\ (\mathbf{r}_\mathrm{s}+1)) \\
s' & = & \textbf{unreachable by}\ (\mathsf{FALSEI}\ (\mathsf{ADDRTL}\ h_s\ h_L) \\
 & & \qquad\qquad\qquad\qquad h_{unsafe}) \\
I'' & = & I_4, \mathbf{r}_\mathrm{pc} = 5, \mathbf{r}_\mathrm{s} = (\mathsf{sel}\ \mathbf{r}_\mathrm{m}\ (\mathbf{r}_\mathrm{s}+1)), \\
 & & h_{safe} : \mathsf{Addr}\ (\mathbf{r}_\mathrm{s}+1) \\
s'' & = & \textbf{abstract}\ (\mathsf{sel}\ \mathbf{r}_\mathrm{m}\ (\mathbf{r}_\mathrm{s}+1))\ \textbf{as}\ t\ \textbf{in} \\
 & & \textbf{set}\ \mathbf{r}_\mathrm{s} = t\ \textbf{by}\ \mathsf{ID}\ \textbf{in} \\
 & & \textbf{assert}\ h_s : \mathsf{List}\ t\ \textbf{by}\ (\mathsf{RDTL}\ h_s\ h_L)\ \textbf{in} \\
 & & \textbf{forget}\ h_{safe}\ \textbf{in} \\
 & & \textbf{collect as}\ I_5
\end{array}$$

These scripts require the introduction of two new proof rules, which can be justified using the definitions of Nelist and LInv introduced in Section 2.2:

$$\begin{array}{l}
\mathsf{ADDRTL} : \Pi m{:}\mathsf{mem}.\ \Pi\ell{:}\mathsf{val}.(\mathsf{Nelist}\ \ell) \rightarrow (\mathsf{LInv}\ m) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{Addr}\ (\ell+1))
\end{array}$$

$$\begin{array}{l}
\mathsf{RDTL} : \Pi m{:}\mathsf{mem}.\ \Pi\ell{:}\mathsf{val}.(\mathsf{Nelist}\ \ell) \rightarrow (\mathsf{LInv}\ m) \rightarrow \\
\qquad\qquad\qquad\qquad (\mathsf{List}\ (\mathsf{sel}\ m\ (\ell+1)))
\end{array}$$

The script s' shows that the invariant I' is unreachable by deriving falsehood using the built-in FALSEI proof rule, which takes as arguments a proposition (here left implicit), its proof, and a proof of its negation.

The script s'' starts by abstracting the value of $\mathbf{r}_\mathrm{s}$ with a new existential variable t. Then we replace the old assumption about $\mathbf{r}_\mathrm{s}$ with an assumption about the new value of $\mathbf{r}_\mathrm{s}$. Finally, we forget the assumption that the address was safe to read.

Consider now the processing of the jump instruction from line 6 in Figure 3. The extension notices that this instruction closes a loop, and it attempts to verify whether the invariant already collected for the loop head (I_2) is weaker than the current invariant. This is the case, and the extension produces the script shown below:

$$\begin{array}{rcl}
I_7 & = & \mathbf{r}_\mathrm{pc} = 7, h_L : \mathsf{LInv}\ \mathbf{r}_\mathrm{m},\ h_s : \mathsf{List}\ \mathbf{r}_\mathrm{s},\ h_c : \mathsf{Nelist}\ \mathbf{r}_\mathrm{c} \\
I' & = & I_7, \mathbf{r}_\mathrm{pc} = 2 \\
s' & = & \textbf{assert}\ h_c : \mathsf{List}\ \mathbf{r}_\mathrm{c}\ \textbf{by}\ \mathsf{NELISTLIST}\ h_c\ \textbf{in} \\
 & & \textbf{match}\ I_2
\end{array}$$

[6] We allow inferable arguments to be implicit, so that we need not write ($\mathsf{NELIST}\ \mathbf{r}_\mathrm{s}\ h_s\ h_{branch}$).

After the call to `cons`, we will know that $\mathbf{r}_\mathrm{c}$ is in fact of type Nelist; however, the loop invariant is just that $\mathbf{r}_\mathrm{c}$ is a List. We use the (obvious) lemma NELISTLIST to forget that $\mathbf{r}_\mathrm{c}$ is non-empty:

$$\mathsf{NELISTLIST} : \Pi\ell{:}\mathsf{val}.\ (\mathsf{Nelist}\ \ell) \rightarrow (\mathsf{List}\ \ell)$$

Note that the **assert** is not to be understood as recursive, but as imperatively replacing the assumption named h_c with a new one. Afterward, we have something equivalent to the already collected I_2.

Some extensions are not able to provide the loop invariant when the loop is first encountered (e.g., dataflow based extensions) and need to process the loop body several times, weakening the loop invariant. The most natural way to write such an extension is to make it collect weaker and weaker invariants, with the Open Verifier processing the body several times. Alternatively, the extension may want to compute the fixed-point ahead of time and then have the Open Verifier iterate only once.

2.5 The Trusted Checker Module

The Checker module is responsible for interpreting and checking the scripts returned by the extensions. We model the interpretation of scripts using the judgment:

$$\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash s \rightarrow \Delta'$$

which means that, given a set Δ of named invariants already collected, script s results in the collection Δ', along with a proof that invariant I is stronger than the disjunction of invariants in $\Delta \cup \Delta'$. The substitution Σ is a finite map from existential variables to expressions and keeps track, for the duration of the script, of the concrete values of the newly introduced existential variables. Initially, Σ is empty.

We use the notation I_Γ and I_Ψ for the context and respectively the assumptions component of invariant I; the notation $\Sigma(e)$, $\Sigma(\phi)$, and $\Sigma(\Psi)$ to apply the substitution on expressions, formulas, and lists of assumptions, respectively; and the notation $I \equiv I'$ to indicate that I and I' are equivalent up to alpha-renaming of existential variables and dropping of unused existential variables from the context. Figure 6 shows the rules for the interpretation of scripts. These rules make use of two additional judgments:

$$\begin{array}{c}
\Gamma \vdash e : \tau \\
\Gamma, \Psi \vdash \gamma : \phi
\end{array}$$

for type checking that e has type τ in context Γ, and for checking that γ is a representation of a proof of ϕ, in context Γ and assumptions Ψ. These judgments come from an underlying logic, which we chose to be the Calculus of Inductive Constructions [CH85] in our implementation.

We have proved precise soundness properties of these rules. Intuitively, the rules correspond with standard natural deduction proof rules, as follows:

ABSTRACT	existential introduction
SET	substitution of equivalent terms
ASSERT	conjunction introduction
FORGET	conjunction elimination
UNREACHABLE	false elimination
CASES	disjunction elimination
MATCH	hypothesis rule

$$\frac{I_\Gamma \vdash e : \tau \quad \Delta \mathbin{;} I, x : \tau \mathbin{;} \Sigma, x = \Sigma(e) \vdash s \to \Delta' \quad x \notin I_\Gamma}{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{abstract } e \textbf{ as } x \textbf{ in } s \to \Delta'} \text{ ABSTRACT} \qquad \frac{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash s \to \Delta'}{\Delta \mathbin{;} I, h : \phi \mathbin{;} \Sigma \vdash \textbf{forget } h \textbf{ in } s \to \Delta'} \text{ FORGET}$$

$$\frac{I_\Gamma, \Sigma(I_\Psi) \vdash \gamma : \Sigma(\phi) \quad \Delta \mathbin{;} I, h : \phi \mathbin{;} \Sigma \vdash s \to \Delta'}{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{assert } h : \phi \textbf{ by } \gamma \textbf{ in } s \to \Delta'} \text{ ASSERT} \qquad \frac{I_\Gamma, \Sigma(I_\Psi) \vdash \gamma : (\Sigma(e) = \Sigma(e')) \quad \Delta \mathbin{;} I, r = e' \mathbin{;} \Sigma \vdash s \to \Delta'}{\Delta \mathbin{;} I, r = e \mathbin{;} \Sigma \vdash \textbf{set } r = e' \textbf{ by } \gamma \textbf{ in } s \to \Delta'} \text{ SET}$$

$$\frac{I_\Gamma, \Sigma(I_\Psi) \vdash \gamma : \bot}{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{unreachable by } \gamma \to \emptyset} \text{ UNREACHABLE} \qquad \frac{I_\Gamma, \Sigma(I_\Psi) \vdash \gamma : \phi_1 \vee \phi_2 \quad \begin{array}{c} \Delta \mathbin{;} I, h_1 : \phi_1 \mathbin{;} \Sigma \vdash s_1 \to \Delta_1 \\ \Delta \mathbin{;} I, h_2 : \phi_2 \mathbin{;} \Sigma \vdash s_2 \to \Delta_2 \end{array}}{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{cases } \gamma \textbf{ of } h_1 \Rightarrow s_1 \mid h_2 \Rightarrow s_2 \to \Delta_1 \cup \Delta_2} \text{ CASES}$$

$$\frac{}{\Delta \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{collect as } u \to \{u : I\}} \text{ COLLECT} \qquad \frac{I \equiv I'}{\Delta, u : I' \mathbin{;} I \mathbin{;} \Sigma \vdash \textbf{match } u \to \emptyset} \text{ MATCH}$$

Figure 6: The operational semantics of scripts.

2.6 The Trusted Fixpoint Module

Let *Extension*(I') denote the script returned by the extension on the post-invariant I'. Let *Checker*(s, Δ, I') be Δ' where $\Delta \mathbin{;} I' \mathbin{;} \cdot \vdash s \to \Delta'$.

We define Fixpoint by means of the *Step* function as follows:

$$\mathit{Step}(\Delta, I) = \Delta \cup \bigcup_{I' \in \mathrm{Post}(I)} \mathit{Checker}(\mathit{Extension}(I'), \Delta, I')$$

Our implementation uses a worklist algorithm to successively apply *Step* to members of a growing Δ set until a fixed point is reached. Here we will work with a simpler model that nondeterministically chooses a member of the current Δ, applies *Step* to it, adds the output to Δ, and loops until this operation does not grow Δ for any choice of a member to expand.

Formally, the Fixpoint module starts with a set of invariants containing just the initial invariant I_0, and expands it using the *Step* function until it obtains a Δ with the following properties (corresponding directly with Figure 1):

- $I_0 \in \Delta$, and
- For each $I_i \in \Delta$, $\mathit{Step}(\Delta, I_i) \subseteq \Delta$

This is a standard fixed point condition for an abstract interpretation. Note that all the interesting work of the verification, creating invariants and producing proofs, is up to the extension. The Fixpoint module just collects the invariants and sends the proofs to be checked, and when the extension only refers to old invariants by name rather than creating new invariants, it determines that the verification is complete.

It is not possible to prove in general that this process terminates. It is the job of the extension to ensure this, by abstracting information and weakening concrete states as necessary. Thus, while a concrete exploration of the state space may have no fixed point Δ, extensions provide abstractions that identify sufficiently similar states and permit the algorithm to terminate. More coarse-grained abstractions can lead to smaller final Δ and correspondingly faster convergence.

There are potential concerns arising from our decision to query arbitrary, untrusted extension programs during verification. Our soundness theorem requires that extensions act as "black boxes" that can't interfere with the trusted infrastructure's data structures or termination. However, an arbitrary binary extension may itself violate the safety policy, in which case we can't guarantee the needed properties. Possible solutions include running the extension in a sandbox or requiring that the extension first be certified safe by another extension (where we bootstrap with one sandboxed root extension for some very simple safety mechanism). Another potential concern is that extensions may launch "denial of service attacks", either by going into infinite loops when queried, or by always collecting new invariants, preventing a fixed point from being reached. However, this issue is not really specific to our approach. All certified code techniques allow code to be distributed with proofs, invariants, or types that are large enough to prolong verification far beyond acceptable thresholds. In the setting of the Open Verifier, we can let the code consumer set, or negotiate with extensions, policies on how long extensions are permitted to take per step and in total, aborting any verifications that exceed these parameters. A benevolent extension can always arrange with the code producer to attach additional metadata to the code, to enable it to produce the scripts quickly enough.

Though we omit it here due to lack of space, we have proved the soundness of this formalism. The proof is essentially an application of standard reasoning about abstract interpretation. Together with straightforward proofs showing that each operation of Post corresponds to an axiomatization of machine semantics, a formalization of the soundness proof could be used to compile an Open Verifier verification into pure FPCC.

2.7 Proof Layering

A key part of the Open Verifier design for simplifying the job of the extension writer is a particular layered approach to proof construction. The invariants are existentially quantified conjunctions of formulas, which, in almost all cases in the extensions we have written, are atomic. Thus most reasoning can be restricted to Horn logic, which is easily automatizable. Our implementation includes an untrusted proof-generating Prolog interpreter, which can be used as a library by extensions. (In type-based extensions, Prolog can be used as a foundational type checker, where the typing rules are implemented as a Prolog program.) We have found all program-specific reasoning can be handled this way.

The extension's formula constructors, and the lemmas relating them that are used as Prolog rules, must be implemented in a richer logic. Our implementation uses the

Coq [Coq02] proof assistant for the Calculus of Inductive Constructions [CH85], but in fact all of our experiments, even those like TAL which manipulate function pointers directly, require only first-order reasoning together with inductively defined types and predicates. This proof layer is for extension-specific (which usually means compiler-specific) reasoning.[7]

Finally there is the soundness of the trusted Open Verifier framework itself, which incorporates elements common to all verifiers and needs to be proven only once. This proof either requires higher-order logic or needs to be interpreted as a first-order proof schema over finite collections of invariants.

Besides producing the necessary weakening proofs, extensions also must produce the invariants. Often, we have found it fruitful to start with some notion of an abstract state at each point, which is manipulated by a conventional non-proof-producing verifier; the extension then translates the abstract states into invariants for the Open Verifier and produces the needed proofs. This is similar in spirit to [SYY03], where Hoare logic proofs are automatically produced from the results of abstract interpretations; they automatize proof production in a very generic way (as opposed to our extension-specific proof production strategies), but they also work at the source-code level for a very restricted source language.

3. CASE STUDIES

To explore the flexibility of the Open Verifier, we have implemented extensions with a large variance in abstraction, inference, and language features.

3.1 Proof-Carrying Code

In this section, we consider a verification-condition generator similar to the one used in Touchstone [CLN+00] as an untrusted extension for the Open Verifier. This is perhaps the simplest extension of all. The script for almost any instruction is simply a **collect**, indicating the desire to use strongest postconditions. For loop entry points, the code metadata declares loop-modified registers and invariant predicates. These are translated directly into abstractions for the modified registers, along with assertions for the invariant predicates, followed by a **collect** the first time the loop is encountered, or a **match** the subsequent times.

The proofs required for the **assert** scripts are extracted from the metadata and re-packaged as needed by the Checker. It was a liberating feeling when compared to previous implementations of PCC that we do not need to trust the verification-condition generator anymore[8] and that we do not have to standardize a proof-representation format. It is all between the PCC producer and the writer of the extension.

[7] This proof layering is similar to that proposed by [WAS03], which incorporates Prolog into the trusted proof checker instead of using an untrusted proof-generating Prolog interpreter.

[8] Recall that our trusted postcondition generator is quite far from the complexity of traditional VCGens. (Compare the less than 3000 lines of trusted code in our implementation with the over 6000 lines in Touchstone's [CLN+00] VCGen alone.) It does not deal with loop invariants or any compilation strategy-specific mechanisms, but just directly reflects the machine semantics.

Similar work on implementing a foundational Hoare-logic-style verifier is described in [YHS03, HS04]. Like those authors, we are interested in using low-level verifiers to verify foundationally the run-time systems of high-level languages. This would involve the joint operation of the low-level verifier with a high-level, probably type-based, verifier to produce a complete foundational verification of a given piece of untrusted code. In [HS04], this requires a re-expression of their high-level type-based FPCC system, originally conceived using a global well-typedness invariant, in terms of the local state predicates holding at interfaces with the low-level system. We believe our system offers an advantage in having all verifiers already expressed in terms of local invariants which are state predicates. Producing a complete verification of a realistic run-time system is still a work in progress.

3.2 Typed Assembly Language

In this section, we consider an Open Verifier extension for verifying programs generated from x86 Typed Assembly Language (TALx86) [MCG+99]. The extension is built around existing compilers and checkers distributed in the 1.0 release of the TALC tools from Cornell University [MCG+03]. This TAL version includes stack, product, sum, recursive, universal, existential, array, and continuation types. The operations on values of all of these types, including dynamic memory allocation, are handled soundly from first principles by our TAL extension. We are able to verify the results of compiling the test cases included in TALC for its compilers for Popcorn (a safe C dialect) and mini-Scheme.

We were able to use the standard TALC distribution's type system, compilers, and type checker unchanged in the extension, without requiring trust in any of these components. The extension uses the distributed TALx86 type checker to produce abstract TAL states at each program point, which are then translated into state predicates for the Open Verifier. A little extra work must be done to produce invariants for instructions in the middle of TALx86 macros like memory allocation; currently, we use a simple non-garbage-collected allocator for which this is rather straightforward, although we are exploring more realistic run-time functions. After the invariants are created, the needed weakening proofs are produced automatically using lemmas proven by hand in Coq.

The majority of our implementation effort involved identifying and proving the lemmas that underlie informal proofs of the type checker's correctness. However, instead of doing this by proving progress and preservation properties of an abstract semantics, we assign direct meanings to typing predicates in terms of concrete machine states. The extension uses typing information produced by the checker to determine which lemmas are needed to produce individual proofs of any new assumptions that it asserts. We have found that this decentralized proof strategy eases the extension writer's burden to an extent by reducing the need for "congruence rules," but more experience is needed to say anything concrete.

The reader may at this point find himself perplexed as to how we handle verification based on a thoroughly "higher order" mechanism like TAL using only a subset of first-order logic. In fact, our previous work on the Open Verifier [NS03, Sch04] relied on a more "higher order" notion of invariant corresponding to natural-number-indexed state predicates, in a way comparable with [AM01].

However, in the TAL extension, we use a formulation where we define the "value e has code type τ" predicate to mean "e is one of the fixed set of code block addresses whose types are subtypes of τ". Subtyping is handled through first-order reasoning about syntactic substitution for universally quantified type variables. Therefore, a jump can be proven safe by a case analysis on all possible targets of compatible type. By the construction of our invariants, it is easy to convert the source invariant to **match** the target invariant in each case. To avoid the overhead of a case analysis at each jump, we use an invariant that is satisfied by any valid jump target state and prove it safe with a single case analysis at the start of verification. Further details on the TAL extension be found in [Chl04].

3.3 The Dataflow Based Extension for Cool

We have also built a verifier for a type-safe object-oriented programming language called Cool (Classroom Object-Oriented Language [Aik96])—more precisely, for the assembly code produced by a broad class of Cool compilers. The most notable features of Cool are a single-inheritance class hierarchy, a strong type system with subtyping, dynamic dispatch, a type-case construct, exceptions, and self-type polymorphism. For our purposes, it can be viewed as a realistic subset of Java or C#. We chose Cool for one of our case studies because it tests the flexibility and usability of the Open Verifier for making a JVML-like verifier untrusted, and it tests the ability to retrofit FPCC to existing compilers and languages.

Other efforts in FPCC systems for Java-like languages have assumed a type-preserving compilation to a general (if not universal) typed assembly language on which the code is certified [LST02]. In this study, we consider the case when it is infeasible or impractical to develop such a compiler. Furthermore, we assert that such an encoding of object-oriented languages in traditionally functional TALs can be unnatural, much like the compilation of functional languages to the JVML or CIL; others seem to concur [CT05]. A design decision in [LST02] to change the compilation scheme of the type-case rather than introduce a new tag type (which they found possible but difficult in their system) provides some additional evidence for this.

To summarize our results, our Cool extension today indeed verifies the safety of programs generated by a compiler that we have left unchanged throughout this process. In fact, our success in achieving compiler re-use is underscored by the fact that the Cool extension has been quite successful in proving the safety of the output of dozens of Cool compilers developed by students in the undergraduate compilers class at UC Berkeley. Further detail about our verification strategy and experiments is available in a separate paper [CCNS05].

4. IMPLEMENTATION

We have an implementation of the Open Verifier framework in OCaml, along with a graphical interactive user interface. Moreover, we have built several extensions that demonstrate the Open Verifier's ability to support a wide range of verification strategies, including an extension for traditional PCC called PCCExt (Section 3.1), one for TALx86 called TALExt (Section 3.2), an extension for compiled-Cool called Coolaid (Section 3.3), and one for the example language L called LExt (Section 2.1). We have used the implementation to verify programs from source code written in Cool; Popcorn and mini-Scheme (for TALExt); and C (compiled with `gcc -fstack-check`, for PCCExt and LExt). Our examples are up to about 600 lines of source code, and we are confident that there is no obstacle to verifying larger programs.

	Layer	PCCExt lines	TALExt lines	Coolaid lines
Untrusted	Conventional	–	6,000	3,600
	Wrapper	2,400	3,300	3,300
	Total	**2,400**	**9,300**	**6,900**
	Prolog Rules	200	1,600	900
	Coq Proofs	300	17,500	4,000

Table 1: Size of the extensions.

In our implementation, Post operates on a generic untyped RISC-like assembly language called SAL; we have parsers that translate both x86 and MIPS assembly languages into SAL. Representing SAL takes 300 lines of code, while the MIPS and x86 parsers are 700 and 1,000 lines, respectively. The implementation of the trusted modules `Post` (100 lines), `Checker` (600 lines for script interpretation along with 500 lines for proof checking), and `Fixpoint` (200 lines) follow directly from their descriptions in Sections 2.3, 2.5, and 2.6, respectively. Thus, the amount of trusted code in these modules is either 2400 lines of OCaml if you use MIPS, or 2700 for Intel x86. This number does not include the size of the Coq kernel (8000 lines), which we use at the moment to check the proofs of lemmas. However, since we do not need the full capabilities even of the Coq kernel, we might be able to use instead a slightly expanded version of the Checker module. In a traditional FPCC trusted base, we see analogues of `Checker` (a trusted proof checker) and `Post` (a logical formalization of machine semantics), but no analogue of `Fixpoint`. This highlights the main difference in the trust relationships that our system requires.

The extensions are the most complex pieces to build. As we have advocated, this is an important reason for making this part untrusted. Though a fair amount of effort is required for a complete extension, development is typically not from scratch and can be staged depending on how much one wants to trust. We foresee most extensions being built upon conventional non-proof-producing verifiers; the applicability of this approach is confirmed especially by TALExt where the conventional verifier, TALx86, was not built or modified by us. The cost above constructing conventional verifiers is made manageable using the extension scripts described in Section 2.4.1. At this point, one may be satisfied with trusting the lemmas about the enforcement mechanism (e.g., the typing rules along with their soundness lemma), but we no longer need to trust their implementation in the conventional verifier. However, if this is unsatisfactory, proofs of these lemmas can be formalized in a machine checkable form, as we have done for Coolaid (in part) and for TALExt using Coq [Coq02].

Furthermore, extensions can often share modules handling common software abstractions. For example, both Coolaid and LExt share modules for handling stacks, functions, and memory regions. All extensions utilize to some

Extension	Conv.	Prolog	Checking	Other
TAL small	0.00	0.03	0.01	0.05
TAL medium	0.01	0.05	0.02	0.10
TAL large	0.21	2.91	0.74	2.94
Cool small	0.01	0.00	0.02	0.11
Cool medium	0.02	0.37	0.15	0.30
Cool large	0.22	3.98	1.61	4.59

Table 2: Running times (in seconds) for selected verifications on a 1.7 GHz Pentium 4.

degree the Prolog interpreter for automating the proving of per-program facts in terms of lemmas (i.e., for serving as a customizable type-checker); however, proofs can be constructed directly without using Prolog, say for efficiency reasons (as we do for the simpler LExt). The size of the implementations of the major extensions—PCCExt, TALExt, and Coolaid—are given in Table 1. In all cases except for PCC, there is more untrusted code than trusted. Also, we see that the size of a wrapper is smaller than that of a conventional verifier.

We have tested our implementations on suites of test programs. Here we present results for a representative range of inputs. For both TAL and Cool, Table 2 and Figure 7 present a breakdown of running time on selected examples ranging from small "hello world" programs, to medium-sized cases with a few hundred assembly instructions, to large cases with around 6000 instructions. Running time is broken into 4 categories: the time taken by the conventional verifier, the Prolog prover, the trusted proof checker, and any other parts of the system. The last category includes overhead from both the trusted parts of the Open Verifier and from the extension's own bookkeeping. We see that both naive proof generation and checking take up significant portions of time. We expect to be able to improve proof checking drastically by using engineering solutions such as those described in [NL97]. Preliminary experiments lead us to believe that overhead over the time of a conventional verifier can be enormously reduced through careful hand optimization of proof generation, but we have not yet constructed any complete verifiers in this style.

5. CONCLUSION

We have described the Open Verifier framework, which produces foundational verifiers using untrusted extensions to customize the safety enforcement mechanism. We have proved its soundness (i.e., that unsound extension actions will be caught), and we have demonstrated its flexibility for a broad set of verification systems ranging from Hoare-style verifiers to traditional typed assembly languages to dataflow or abstract interpretation based verifiers.

Our framework enables an approach for adding layers of increasing trust in the construction of a verifier. We provide support for starting with a conventional verifier, wrapping it with an extension that applies trusted lemmas using a specialized script language, and then proving those lemmas to arrive at a completely foundational verifier. This layering is also a layering of logical complexity, and in particular in the amount of specialized background in formal methods required to produce the various parts of a foundational verifier.

The Open Verifier provides a standard framework for the production of multiple FPCC instantiations, each using those techniques most natural to a given source language and compilation strategy. It allows the re-use of existing compilers and non-foundational verifiers. Moreover, we have designed the framework for accessibility. In all, we believe our perspectives embodied in the Open Verifier provide a more feasible approach to foundational proof-carrying code. We conjecture that our system supports practical verifiers for almost all safety policies of interest in practice, though it remains for further research to test this hypothesis.

Acknowledgments. We would like to thank Kun Gao for his hard work on the implementation of our Open Verifier prototype and Andrew Appel, Jeremy Condit, Simon Goldsmith, Matt Harren, Scott McPeak, Simon Ou, Valery Trifonov, Wes Weimer, Dinghao Wu, and the anonymous referees for reviewing and providing helpful comments on drafts of this paper.

6. REFERENCES

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, January 2000.

[Aik96] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, July 1996.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. Type-based verification of assembly language for compiler debugging. In *Proceedings of the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, January 2005.

[CH85] Thiery Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL'85), LNCS 203*, pages 151–184. Springer-Verlag, 1985.

[Chl04] Adam Chlipala. An untrusted verifier for typed assembly language. M.S. Report UCB/ERL M04/41, EECS Department, University of California, Berkeley, 2004.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.

[Coq02] Coq Development Team. The Coq proof assistant reference manual, version 7.3. May 2002.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-03)*, volume 38(1) of *ACM SIGPLAN Notices*, pages 198–212. ACM Press, January 15–17 2003.

[CT05] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL-05)*, January 2005.

[Gou02] John Gough. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, Upper Saddle River, New Jersey, 2002.

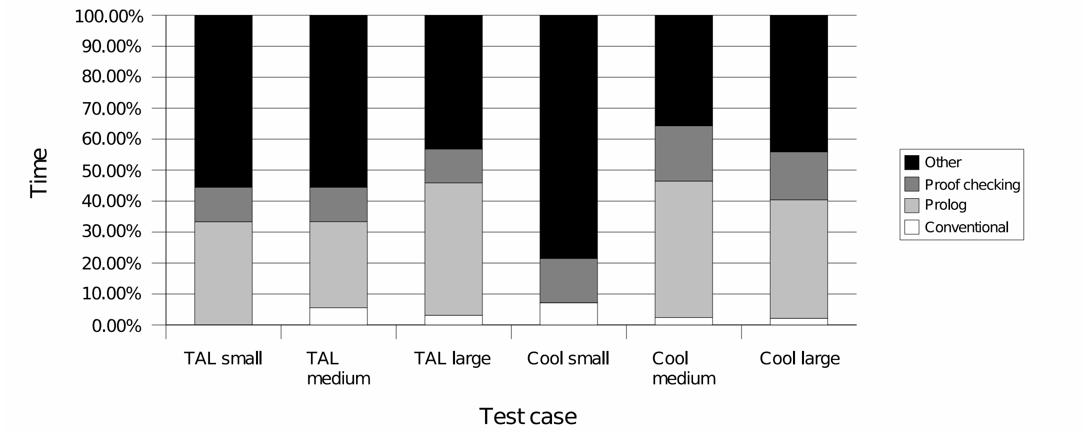

Figure 7: Breakdown of extension execution time (graph of Table 2).

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, pages 248–260, London, United Kingdom, January 2001.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HS04] Nadeem A. Hamid and Zhong Shao. Interfacing Hoare logic and type systems for Foundational Proof-Carrying Code. In *17th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs2004)*, September 2004.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[LST02] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, 2002.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000.

[MCG+99] 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. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[MCG+03] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talc releases, 2003. URL: http://www.cs.cornell.edu/talc/releases.html.

[ML71] Per Martin-Löf. A theory of types. Technical Report 71–3, Department of Mathematics, University of Stockholm, 1971.

[NL97] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University, October 1997.

[NR01] George C. Necula and Shree P. Rahul. Oracle-based checking of untrusted programs. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, pages 142–154. ACM Press, January 2001.

[NS03] George C. Necula and Robert R. Schneck. A sound framework for extensible untrusted verification-condition generators. In *Proceedings of the Eighteenth Annual IEEE Symposium on Logic in Computer Science*, pages 248–260, Ottawa, Canada, June 2003.

[Sch04] Robert R. Schneck. *Extensible Untrusted Code Verification*. PhD thesis, University of California, Berkeley, May 2004.

[SYY03] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *LNCS*. Springer-Verlag, 2003.

[WAS03] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, August 2003.

[YHS03] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proceedings of the 2003 European Symposium on Programming (ESOP'03)*, April 2003.