# A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety*

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula

University of California, Berkeley, California, USA
{bec,adamc,necula}@cs.berkeley.edu

**Abstract.** A *certified program analysis* is an analysis whose implementation is accompanied by a checkable proof of soundness. We present a framework whose purpose is to simplify the development of certified program analyses without compromising the run-time efficiency of the analyses. At the core of the framework is a novel technique for automatically extracting Coq proof-assistant specifications from ML implementations of program analyses, while preserving to a large extent the structure of the implementation. We show that this framework allows developers of mobile code to provide to the code receivers untrusted code verifiers in the form of certified program analyses. We demonstrate efficient implementations in this framework of bytecode verification, typed assembly language, and proof-carrying code.

## 1 Introduction

When static analysis or verification tools are used for validating safety-critical code [BCC+03], it becomes important to consider the question of whether the results of the analyses are trustworthy [KN03,BCDdS02]. This question is becoming more and more difficult to answer as both the analysis algorithms and their implementations are becoming increasingly complex in order to improve precision, performance, and scalability. We describe a framework whose goal is to assist the developers of program analyses in producing formal proofs that the implementations and algorithms used are sound with respect to a concrete semantics of the code. We call such analyses *certified* since they come with machine-checkable proofs of their soundness. We also seek soundness assurances that are *foundational*, that is, that avoid assumptions or trust relationships that don't seem fundamental to the objectives of users. Our contributions deal with making the development of such analyses more practical, with particular emphasis on not sacrificing the efficiency of the analysis in the process.

The strong soundness guarantees given by certified program analyzers and verifiers are important when the potential cost of wrong results is significant.

* This research was supported in part by NSF Grants CCR-0326577, CCF-0524784, and CCR-00225610; an NSF Graduate Fellowship; and an NDSEG Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

This technical report is an extended version of the manuscript appearing in VMCAI 2006, LNCS 3855.

Moreover, the ability to check independently that the implementation of the analysis is sound allows us to construct a mobile-code receiver that allows untrusted parties to provide the code verifier. The code verifier is presented as a certified program analysis whose proof of soundness entails soundness of code verification.

The main contributions of the framework we propose are the following:

- We describe a methodology for translating automatically implementations of analyses written in a general-purpose language (currently, ML) into models and specifications for a proof assistant (currently, Coq). Specifically, we show how to handle those aspects of a general-purpose language that do not translate directly to the well-founded logic used by the proof assistant, such as side-effects and non-primitive recursive functions. We use the framework of abstract interpretation [CC77] to derive the soundness theorems that must be proved for each certified analysis.
- We show a design for a flexible and efficient mobile-code verification protocol, in which the untrusted code producer has complete freedom in the safety mechanisms and compilation strategies used for mobile code, as long as it can provide a code verifier in the form of a certified analysis, whose proof of soundness witnesses that the analysis enforces the desired code-receiver safety policy.

In the next section, we describe our program analysis framework and introduce an example analyzer. Then, in Sect. 3, we present our technique for specification extraction from code written in a general-purpose language. We then discuss the program analyzer certification process in Sect. 4. In Sect. 5, we present an application of certified program analysis to mobile code safety and highlight its advantages and then describe how to implement in this architecture (foundational) typed assembly language, Java bytecode verification, and proof-carrying code in Sect. 6. Finally, we survey related work (Sect. 7) and conclude (Sect. 8).

This technical report is an expanded version of a paper presented at VMCAI 2006 [CCN06]. The main additions in this paper are as follows:

- in Sect. 2.1, an expanded description of the Java bytecode verifier-like example, including new or expanded figures 4, 5, and 6;
- in Sect. 3.2, a formalization of our specification extraction technique for a mini-ML language, including a proof of the main soundness theorem;
- in Sect. 4, new subsections sketching a proof of soundness for our certified program analysis framework and the local soundness proofs for the running bytecode verifier-like example; and
- in Sect. 5, a new figure that highlights the differences between our certified verifier architecture and traditional proof-carrying code implementations (Fig. 13).

## 2 The Certified Program Analysis Framework

In order to certify a program analysis, one might consider proving directly the soundness of the implementation of the analysis. This is possible in our frame-

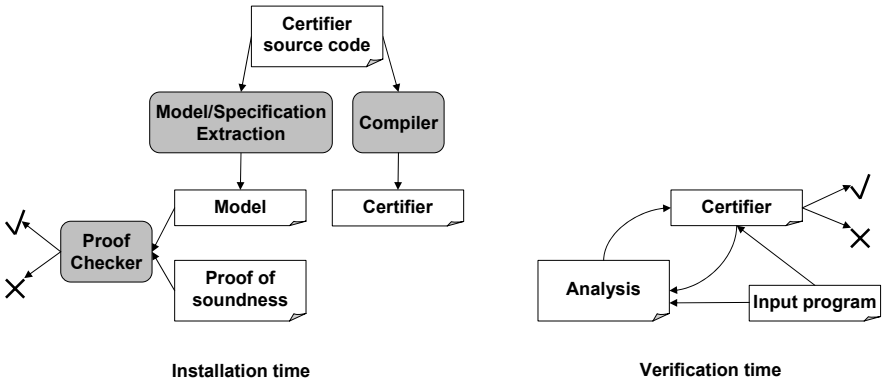


**Fig. 1.** Our certified verifier architecture with the trusted code base shaded

```
type absval
type abs = { pc : nat; a : absval }
val ainv : abs list
val astep : abs -> result
datatype result = Fail | Succ of abs list
```

**Fig. 2.** The core types of a certifier

work, but we expect that an alternative strategy is often simpler. For each analysis to be certified, we write a certifier that runs after the analysis and checks its results. Then, we prove the soundness of the certifier. This approach has several important advantages. Often the certifier is simpler than the analysis itself. For example, it does not need to iterate more than once over each instruction, and it does not need all the complicated heuristics that the analysis itself might use to speed up the convergence to a fixpoint. Thus, we expect the certifier is easier to prove sound than the analysis itself. The biggest benefit, however, is that we can use an existing implementation of a program analysis as a black box, even if it is written in a language that we are not ready to analyze formally, and even if the analysis algorithm does not fit perfectly with the formalism desired for the certification and its soundness proofs. As an extreme example, the analysis itself might contain a model checker, while we might want to do the soundness proof using the formalism of abstract interpretation [NJM+02]. In Fig. 1, we diagram this basic architecture for the purpose of mobile-code safety. We distinguish between "installation time" activity, which occurs once per analyzer, and "verification time" activity, which occurs once per program to analyze.

We choose the theory of abstract interpretation [CC77] as the foundation for the soundness proofs of certifiers because of its generality and because its soundness conditions are simple and well understood. We present first the requirements for the developers of certifiers, and then in Sect. 4, we describe the soundness verification.

```
fun applyWithTimeout (f: 'a -> 'b, x: 'a) : 'b = ...
fun top (DonePC: nat list, ToDo: abs list) : bool =
 case ToDo of
   nil => true
 | a :: rest =>
      if List.member(a.pc, DonePC) then false else
      (case applyWithTimeout(astep, a) of
         Fail => false
       | Succ as => top (a.pc :: DonePC, as @ ToDo))
in
  top (nil, ainv)
```

**Fig. 3.** The trusted top-level analysis engine. The infix operators :: and @ are list cons and append, respectively

The core of the certifier is an untrusted custom module containing an implementation of the abstract transition relation (provided by the certifier developer). The custom module of a certifier must implement the signature given in Fig. 2. The type `abs` encodes abstract states, which include a program counter and an abstract value of a type that can be chosen by the certifier developer. The value `ainv` consists of the abstract invariants. They must at a minimum include invariants for the entry points to the code and for each destination of a jump. The function `astep` implements the abstract transition relation: given an abstract state at a particular instruction, compute the set of successor states, minus the states already part of `ainv`. The transition relation may also fail, for example when the abstract state does not ensure the safe execution of the instruction. We will take advantage of this possibility to write safety checkers for mobile-code using this framework. In our implementation and in the examples in this paper, we use the ML language for implementing custom certifiers.

In order to execute such certifiers, the framework provides a trusted engine shown in Fig. 3. The main entry point is the function `top`, invoked with a list of program counters that have been processed and a list of abstract states still to process. Termination is ensured using two mechanisms: each invocation of the untrusted `astep` is guarded by a timeout, and each program counter is processed at most once. We use a timeout as a simple alternative to proving termination of `astep`. A successful run of the code shown in Fig. 3 is intended to certify that all of the abstract states given by `ainv` (i.e., the properties that we are verifying for a program) are invariant, and that the `astep` function succeeds on all reachable instructions. We take advantage of this latter property to write untrusted code verifiers in this framework (Sect. 5). We discuss these guarantees more precisely in Sect. 4.

### 2.1 Example: Java Bytecode Verifier

Now we introduce an example program analyzer that requires the expressivity of a general-purpose programming language and highlights the challenges in specification extraction. In particular, we consider a certifier in the style of the

```
type label
datatype instr =
         RegReg of reg * reg
       | RegLabel of reg * label
       | Write of reg * int * reg
       | Read of reg * int * reg
       | Jump of reg
val instrAt : int -> instr
type class
val fieldOf : class * int -> class option
val super : class -> class option
```

**Fig. 4.** Details of the $\mathcal{J}$ machine.

Java bytecode verifier, but operating on a simple assembly language instead of bytecodes.

Fig. 4 presents the ML definitions that describe this code target, which we'll call the $\mathcal{J}$ machine. Included are the instruction decoder `instrAt`, which maps program counters to the instructions that they reference; the partial function `fieldOf`, which returns the type of a field at a certain offset of a class; and the partial function `super`, which returns the superclass of a class. For simplicity of exposition, we assume that these functions and the object representation convention that they deal with are fixed and part of the trusted code base.

Figs. 5 and 6 present a fragment of this custom verifier. The abstract value is a partial map from registers to types, with a missing entry denoting an uninitialized register. A type is either a class name or a continuation type. Each branch of the `case` expression considers a different assembly instruction. For example, the third branch deals with memory writes. It succeeds only if the destination address is of the form `rdest` + `n`, with register `rdest` pointing to an object of class `cdest` that has at offset `n` a field of type `c′`, which must be a super class of the type of register `rsrc`.

We omit the code for `calculatePreconditions`, a function that obtains some preconditions from the meta-data packaged with the `.class` files and then uses an iterative fixed-point algorithm to find a good typing precondition for each program label. Each such precondition should be satisfied any time control reaches its label. This kind of algorithm is standard and well studied, in the context of the Java Bytecode Verifier and elsewhere, so we omit the details here. Most importantly, we will not need to reason formally about the correctness of this algorithm.

## 3 Specification Extraction

To obtain certified program analyses, we need a methodology for bridging the gap between an implementation of the analysis and a specification that is suitable for use in a proof assistant. An attractive technique is to start with the specification and its proof, and then use *program extraction* supported by proof assistants

```
type ('a, 'b) partialmap
val sel : ('a, 'b) partialmap * 'a -> 'b option
val upd : ('a, 'b) partialmap * 'a * 'b option -> ('a, 'b) partialmap

datatype ty =
       | Class of class
       | Cont of abs
and absval = (reg, ty) partialmap
and abs = { pc : nat; a : absval }

fun subClass (c1 : class, c2 : class) =
    c1 = c2 orelse
    (case super c1 of
         SOME(sup) => subClass(sup, c2)
       | NONE => false)

fun subAbs (a : abs, a' : abs) =
    a.pc = a'.pc andalso
    forall (fn (r, ty) =>
               case sel(a.a, r) of
                   SOME ty' => subType(ty, ty')
                 | NONE => false) a'.a

and subType (ty : ty, ty' : ty) =
    case (ty, ty') of
        (Class c1, Class c2) => subClass(c1, c2)
      | (Cont a1, Cont a2) => subAbs(a2, a1)
      | _ => false

fun calculatePreconditions () : abs list = (* ... *)

val ainv: abs list = calculatePreconditions ()

fun precondition (l : label) : abs = ... (* lookup in ainv *)
```

**Fig. 5.** Support functions for a verifier in the style of the Java Bytecode Verifier

```
fun astep (a : abs) : result =
  case instrAt(a.pc) of
      RegReg(r1, r2) =>
      Succ [ { pc = a.pc + 1; a = upd(a.a, r1, sel(a.a,r2)) } ]
    | RegLabel(r, l) =>
      Succ [ { pc = a.pc + 1; a = upd(a.a, r, SOME (Cont (precondition l))) } ]
    | Write(r1, n, r2) =>
      (case (sel(a.a, r1), sel(a.a, r2)) of
           (SOME(Class c), SOME(t)) =>
             (case fieldOf(c, n) of
                  SOME(c') =>
                  if subType(t, Class(c')) then
                      Succ [ { pc = a.pc + 1; a = a.a } ]
                  else
                      Fail
                | _ => Fail)
         | _ => Fail)
    | Read(r1, r2, n) =>
      (case sel(a.a, r2) of
           Class c =>
           (case fieldOf(c, n) of
                SOME(c') => Succ [ { pc = a.pc + 1; a = SOME (Class c') } ]
              | _ => Fail)
         | _ => Fail)
    | Jump r =>
      (case sel(a, r) of
           SOME(Cont a') =>
           if subAbs(a, a') then
               Succ []
           else
               Fail
         | _ => Fail)
```

**Fig. 6.** Step function of a verifier in the style of the Java bytecode verifier

```
fun subClass (depth : nat, c1 : class, c2 : class) : bool option =
  c1 = c2 orelse
  (case super c1 of NONE => SOME false
   | SOME sup => if depth = 0 then NONE else subClass'(depth-1,sup,c2))
```

**Fig. 7.** Translation of the `subClass` function. The boxed elements are added by our translation

such as Coq or Isabelle [Pau94] to obtain the implementation. This strategy is very proof-centric and while it does yield a sound implementation, it makes it hard to control non-soundness related aspects of the code, such as efficiency, instrumentation for debugging, or interaction with external libraries.

Yet another alternative is based on *verification conditions* [Dij75,FM04], where each function is first annotated with a pre- and postcondition, and the entire program is compiled into a single formula whose validity implies that the program satisfies its specification. Such formulas can make good inputs to automated deduction tools, but they are usually quite confusing to a human prover. They lose much of the structure of the original program. Plus, in our experience, most auxiliary functions in a program analyzer do good jobs of serving as their own specifications (e.g., the `subClass` function).

Since it is inevitable that proving soundness will be sufficiently complicated to require human guidance, we seek an approach that maintains as close of a correspondence between the implementation and its model as possible. For non-recursive purely functional programs, we can easily achieve the ideal, as the implementation can reasonably function as its own model in a suitable logic, such as that of the Coq proof assistant. This suggests that we need a way to handle imperative features, and a method for dealing with non-primitive recursive functions.

### 3.1 Overview of the Main Ideas

In this subsection, we give an informal overview of the main ideas behind our approach. The following subsection formalizes what we present here and proves a key soundness property.

**Handling Recursion.** We expect that all invocations of the recursive functions used during certification terminate, although it may be inconvenient to write all functions in primitive recursive form, as required by Coq. In our framework, we force termination of all function invocations using timeouts. This means that for each successful run (i.e., one that does not time out) there is a bound on the call-stack depth. We use this observation to make all functions primitive recursive on the call-stack depth. When we translate a function definition, we add an explicit argument `depth` that is checked and decremented at each function call. Fig. 7 shows the result of translating a typical implementation of the `subClass` function for our running example. The boxed elements are added by the translation. Note that in order to be able to signal a timeout, the return type of the

```
fun readu16 (s: callstate, buff: int array, idx: int) : int =
 256 * (freshread1 s) + (freshread2 s)

fun readu32 (s: callstate, buff: int array, idx: int) : int =
 65536 * readu16(freshstate3 s,buff,i) + readu16(freshstate4 s,buff,i+2)
```

**Fig. 8.** Translation of a function for reading a 16-bit and 32-bit big-endian numbers from a class file. Original body of `readu16` before translation is $256 * \texttt{buff}[\texttt{i}] + \texttt{buff}[\texttt{i} + 1]$

function is an `option` type. Coq will accept this function because it can check syntactically that it is primitive recursive in the `depth` argument.

This translation preserves any *partial correctness* property of the code. For example, if we can prove about the specification that any invocation of `subClass` that yields `SOME true` implies that two classes are in a subclass relationship, then the same property holds for the original code whenever it terminates with the value `true`.

**Handling Imperative Features.** The function `calculatePreconditions` from Fig. 5 uses I/O operations to read and decode the basic block invariants from the `.class` file (as in the KVM [Ros03] version of Java), or must use an intraprocedural fixed-point computation to deduce the basic block preconditions from the method start precondition (as for standard `.class` files). In any case, this function most likely uses a significant number of imperative constructs or even external libraries. This example demonstrates a situation when the result of complex computations is used only as a *hint*, whose exact value is not important for *soundness* but only for completeness. We believe that this is often the case when writing certifiers, which suggests that a monadic [Wad95] style of translation would unnecessarily complicate the resulting specification.

For such situations we propose a cheaper translation scheme that abstracts soundly the result of side-effecting operations. We describe this scheme informally, by means of an example of functions that read from a Java `.class` file 16-bit and 32-bit numbers, respectively, written in big-endian notation, shown in Fig. 8. Each update to mutable state is ignored. Each syntactic occurrence of a mutable-state access is replaced with a fresh abstract function (e.g., `freshread1`) whose argument is an abstraction of the call-stack state. The call-stack argument is needed to ensure that no relationship can be deduced between recursive invocations of the same syntactic state access. Each function whose body reads mutable state, or calls functions that read mutable state, gets a new parameter `s` that is the abstraction of the call-stack state. Whenever such a function calls another function that needs a call-stack argument, it uses a fresh transformer (e.g., `freshstate3`) to produce the new actual state argument.

This abstraction is sound in the sense that it ensures that nothing can be proved about results of mutable state accesses, and thus any property that we can prove about this abstraction also holds for the actual implementation. If we did not have the call-stack argument, one could prove that each invocation of the

`readu16` function produces the same result, and thus all results of the `readu32` are multiple of 65,537. This latter example also shows why we cannot use the `depth` argument as an abstraction of the call-stack state.

Note that our use of "state" differs from the well-known "explicit state-passing style" in functional programming, where state is used literally to track all mutable aspects of the execution environment. That translation style requires that each function that updates the state not only take an input state but also produce an output state that must be passed to the next statement. In our translation scheme states are only passed down to callers, and the result type of a function does not change.

The cost for the simplicity of this translation is a loss of completeness. We are not interested in preserving all the semantics of input programs. Based on our conjecture that we can refactor programs so that their soundness arguments do not depend on imperative parts, we can get away with a looser translation. In particular, we want to be able to prove properties of the input by proving properties of the translation. We do not need the opposite inclusion to hold.

**Soundness of the Specification Extraction.** We argue here informally the soundness of the specification extraction for mutable state. In our implementation, the soundness of the code that implements the extraction procedure is assumed. We leave for future work the investigation of ways to relax this assumption. First, we observe that each syntactic occurrence of a function call has its own unique `freshstate` transformer. This means that, in an execution trace of the specification, each function call has an actual state argument that is obtained by a unique sequence of applications of `freshstate` transformers to the initial state. Furthermore, in any such function invocation all the syntactic occurrences of a mutable state read use unique `freshread` access functions, applied to unique values of the state parameter. This means that in any execution trace of the specification, each state read value is abstracted as a unique combination of `freshread` and `freshstate` functions. This, in turn, means that for any actual execution trace of the *original program*, there is a definition of the `freshread` and `freshstate` parameters that yields the same results as the actual reads. Since all the `freshread` and `freshstate` transformers are left abstract in the specification, any proof about the specification works with any model for the transformers, and thus applies to any execution trace of the original program.

### 3.2 Formalization

In the rest of this subsection, we illustrate our approach by formalizing it for the simple certifier programming language $\mathcal{L}$ defined in Fig. 9. $\mathcal{L}$ is a standard kind of mini-ML including function, sum, recursive, and reference types. Its only aspects that cannot be translated readily into logic are the imperative reference operations and the potential non-termination of recursive functions. We define a translation that removes these problems while preserving the general form of mostly-functional programs.

$$
\begin{array}{lrcl}
\text{Base Types} & b & & \\
\text{Type Variables} & \alpha & & \\
\text{Types} & \tau & ::= & b \mid \mathbf{unit} \mid \tau \to \tau \mid \tau + \tau \mid \mu\alpha.\, \tau \mid \tau\ \mathbf{ref} \\
\text{Variables} & x, f & & \\
\text{Terms} & e & ::= & () \mid x \mid e\ e \mid (\mathbf{fix}\ f(x:\tau):\tau = e) \\
 & & & \mid \mathbf{inl}_\tau(e) \mid \mathbf{inr}_\tau(e) \\
 & & & \mid (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e \mid \mathbf{inr}(y) \Rightarrow e) \\
 & & & \mid \mathbf{roll}_\tau(e) \mid \mathbf{unroll}(e) \\
 & & & \mid \mathbf{ref}\ e \mid !e \mid e := e \\
\text{Program} & p & ::= & \cdot \mid p, x = e
\end{array}
$$

**Fig. 9.** A simple certifier language $\mathcal{L}$.

In the following, we will assume the standard definition of a typing judgment $\Gamma \vdash e : \tau$, meaning that in context $\Gamma$, a finite map from variable names to types, term $e$ has type $\tau$. The typing rules are entirely standard, so we omit them here. Our translation is defined by a judgment $\Gamma \vdash e \hookrightarrow e'; \Sigma$, which says in typing context $\Gamma$, translate term $e$ into term $e'$ and yield new global bindings $\Sigma$. The new global bindings $\Sigma$ are a list of bindings of the form $x : \tau$. Each such binding corresponds to a new opaque function that abstracts the results of accessing mutable state. The variables free in $e'$ may include those that occurred free in $e$ and those new variables introduced in $\Sigma$, as well as special variables $s$ and *depth*, which we describe below.

**Handling Imperative Features.** We would like to avoid writing programs in a monadic style, or translating them into that form, when all of our uses of mutable state are part of optimizations that do not affect soundness. To achieve this goal, we use a special translation where we introduce a set of opaque functions that abstract the results of accessing mutable state. Each syntactic occurrence of a mutable-state access is replaced with a fresh opaque function whose argument is an abstraction of the call-stack state. The call-stack is needed to ensure that no relationship can be deduced between recursive invocations of the same syntactic state access. This abstraction is sound because it ensures that nothing can be proved about mutable state accesses, and thus any property that we can prove about this abstraction also holds for the actual implementation.

To insure that different invocations of the opaque functions yield different results, these functions take a special argument $s$ of type **callstate** that we ensure is distinct (or more accurately, not provably the same) on all executions of the function. Intuitively, $s$ can be thought of as the call-stack state.

Therefore, we designed our translation to enforce the following property:

> *No relationship can be deduced between the results of two different executions of any imperative expression in the program.*

In other words, we have *compiled imperativeness into undefined but consistent behavior.*

$$\frac{\Gamma \vdash e : \tau \quad f \textbf{ fresh}}{\Gamma \vdash \textbf{ref } e \hookrightarrow f(s); f : \textbf{callstate} \to \tau \textbf{ ref}}\ \textsf{alloc}$$

$$\frac{\Gamma \vdash e : \tau \textbf{ ref} \quad f \textbf{ fresh}}{\Gamma \vdash !e \hookrightarrow f(s); f : \textbf{callstate} \to \tau}\ \textsf{deref}$$

$$\frac{}{\Gamma \vdash e_1 := e_2 \hookrightarrow (); \cdot}\ \textsf{assign}$$

**Fig. 10.** Translation rules for references.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau' \to \tau \\ \Gamma \vdash e_1 \hookrightarrow e_1'; \Sigma_1 \quad \Gamma \vdash e_2 \hookrightarrow e_2'; \Sigma_2 \quad f, \mathit{fail} \textbf{ fresh}\end{array}}{\begin{array}{l}\Gamma \vdash e_1\ e_2 \hookrightarrow \\ \quad \textbf{case unroll}(\mathit{depth}) \textbf{ of inl}(x) \Rightarrow \mathit{fail}(s) \\ \qquad\qquad\qquad\qquad \mid \textbf{inr}(y) \Rightarrow e_1'(y)(f(s))(e_2'); \\ \quad \Sigma_1, \Sigma_2, f : \textbf{callstate} \to \textbf{callstate}, \mathit{fail} : \textbf{callstate} \to \tau\end{array}}\ \textsf{app}$$

$$\frac{\Gamma, f : \tau' \to \tau, x : \tau' \vdash e \hookrightarrow e'; \Sigma}{\begin{array}{l}\Gamma \vdash (\textbf{fix } f(x : \tau') : \tau = e) \hookrightarrow \\ \quad (\textbf{fix } f(\mathit{depth} : \textbf{nat})(s : \textbf{callstate})(x : \tau') : \tau = e'; \Sigma\end{array}}\ \textsf{fun}$$

**Fig. 11.** Translation rules for recursion.

Fig. 10 presents the translation rules for references, which forget the details of imperative execution. We abstract each reference cell allocation and dereference with a fresh opaque function of the appropriate type applied to $s$ (rules alloc and deref). The fresh function variables ensure that distinct syntactic occurrences do not yield provably the same result, and the application to $s$ handles recursive invocations of the same syntactic occurrence. We show in the next section how $s$ is updated at a call site. Since no useful information can be gained from state change, reference cell assignments are erased altogether (rule assign).

**Handling Recursion.** Similar to the translation of imperative features, we introduce another special variable *depth* of type **nat** that represents an upper bound on the remaining function call depth (where $\textbf{nat} \stackrel{\text{def}}{=} \mu\alpha.\ \textbf{unit} + \alpha$). Since the original definition of such a function may not make termination apparent, we change the function to be recursive in *depth* instead of the original argument. We are sure to decrement this argument for each recursive call, yielding a primitive recursive result. To model any given terminating run of a program, we can use, for example, the program's running time as the initial *depth* argument.

Fig. 11 gives the rules for translating possibly non-primitive recursive functions. For function application, we insert a check for sufficient remaining depth (rule app). If the depth is insufficient, then we fail by introducing a fresh opaque function *fail* of the appropriate type, analogous to translation of reference cell

$$\frac{}{\Gamma \vdash () \hookrightarrow (); \cdot}\ \mathsf{unit} \qquad \frac{}{\Gamma \vdash x \hookrightarrow x; \cdot}\ \mathsf{var}$$

$$\frac{\Gamma \vdash e \hookrightarrow e'; \Sigma}{\Gamma \vdash \mathbf{inl}_\tau(e) \hookrightarrow \mathbf{inl}_\tau(e'); \Sigma}\ \mathsf{inl} \qquad \frac{\Gamma \vdash e \hookrightarrow e'; \Sigma}{\Gamma \vdash \mathbf{inr}_\tau(e) \hookrightarrow \mathbf{inr}_\tau(e'); \Sigma}\ \mathsf{inr}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 \hookrightarrow e_1'; \Sigma_1 \\ \Gamma \vdash e \hookrightarrow e'; \Sigma \quad \Gamma, y : \tau_2 \vdash e_2 \hookrightarrow e_2'; \Sigma_2\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1 \mid \mathbf{inr}(y) \Rightarrow e_2 \\ \quad \hookrightarrow \mathbf{case}\ e'\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1' \mid \mathbf{inr}(y) \Rightarrow e_2'; \Sigma, \Sigma_1, \Sigma_2\end{array}}\ \mathsf{case}$$

$$\frac{\Gamma \vdash e \hookrightarrow e'; \Sigma}{\Gamma \vdash \mathbf{roll}_\tau(e) \hookrightarrow \mathbf{roll}_\tau(e'); \Sigma}\ \mathsf{roll} \qquad \frac{\Gamma \vdash e \hookrightarrow e'; \Sigma}{\Gamma \vdash \mathbf{unroll}(e) \hookrightarrow \mathbf{unroll}(e'); \Sigma}\ \mathsf{unroll}$$

**Fig. 12.** Direct translation rules.

allocation and dereferencing. If the depth is sufficient, then we allow recursive calls with one decrement of depth and a fresh state. For function definition, we simply bind additional arguments for *depth* and $s$ (rule fun). We use a standard abbreviation to denote multiple argument, curried **fix**es, and we assume the usual conditions to avoid capture of free variables.

The rules for the remaining term constructors are given in Fig. 12. They simply implement a walk over the structure of a term.

**Soundness of the Specification Extraction.** We can now give the relevant soundness property of this translation. We call a property of terms of our logic *extensional* if it can be defined, for some function term $f$ and constant term $c$, as "the property that holds of all terms $e$ such that $f(e)$ reduces to $c$." This definition makes sense in the context of Coq and any other logic based on typed lambda calculus, and it can be rephrased for other settings.

**Theorem 1.** *If*

1. $\cdot \vdash e : \tau$*;*
2. *Evaluation of* $e$ *with a standard operational semantics terminates within* $N$ *steps;*
3. $\cdot \vdash e \hookrightarrow e'; \Sigma$*; and*
4. *In context* $\Sigma, \mathit{depth} : \mathbf{nat}, s : \mathbf{callstate}$*, extensional property* $P$ *of term* $e'$ *is provable*

*then* $P$ *is also provable of* $e$*.*

*Proof.* Assume that, in $\Sigma, \mathit{depth} : \mathbf{nat}, s : \mathbf{callstate}$, we have a proof of $P$ for $e'$. By a standard cut-elimination lemma that we omit here, for any $x : \tau' \in \Sigma$, we

can derive a proof of $e'[m/x]$, where $\Gamma \vdash m : \tau'$. We can iterate this to replace every variable in $\Sigma$ in this way. We would like to instantiate each variable in a way that gives $e$ and the modified $e'$ the same semantics. We will show that our translation is designed so that this is always possible.

Consider each $x : \tau' \in \Sigma$. We would like to come up with an instantiation of $x$ that has the right type and agrees with $e$'s real execution. We note that the translation inserts each new variable in exactly one position. Consider separately each place where a variable is introduced.

Let $F$ be the set of new variables introduced by the translation. We define the type **callstate** to be the domain of "call stacks." In particular, a **callstate** is a list of elements of $F$. We make the initial state an empty stack.

If $x$ was the $f$ introduced by app, then $\tau' = \textbf{callstate} \to \textbf{callstate}$. Define $x$ to be the function that modifies its argument by adding itself to the top of the call stack.

Suppose $x$ was introduced by rule alloc, so $\tau' = \textbf{callstate} \to \tau''$ **ref** for some $\tau''$. Consider some point during the execution of $e$ when the expression translated to use $x$ is reached. The call stack at this point is a *unique identifier*, meaning that the expression will never again be executed with that call stack.

It is straightforward to see why this is true. First, consider the entire execution of $e$ as a tree of function calls, following a standard eager semantics. We can define a partial order $<$ on states such that $s_1 < s_2$ if $s_1$'s call-stack is a prefix of $s_2$'s (i.e., $s_2$ may be the same as $s_1$ or contain additional nested calls). We observe two properties of the call tree:

**For every node, all its children have greater states.**
By our choice above for the variables introduced at calls, every call pushes a new value onto the stack.

**For every node, all of its children have incomparable states.**
Since each state-modifying variable appears only for one syntactic call, a single stack frame will never be active for multiple executions of such a call. It is necessary to go through some recursion for that to occur.

Since $<$ is a partial order, these facts imply that every stack frame has a unique identifier. Since our particular syntactic occurrence of **ref** $e'$ cannot be executed multiple times for a single stack frame, we have that call stacks provide unique identifiers for its execution instances. Therefore, for each call stack active when this expression is reached, we can define $x$ to map that call stack to the actual reference cell returned at that point in $e$'s execution. For all stack values not covered, we can have $x$ map them to some arbitrary value fixed ahead of time, such as one "dummy reference cell" per type. Our above argument establishes that this uniquely defines $x$'s denotation.

An analogous argument establishes the same result for the variable introduced by deref and the *fail* variable introduced by app.

We have constructed a model for each binding in $\Sigma$ such that, with these instantiations substituted, $e'$ exhibits the same semantics as $e$. Since we can prove property $P$ for this substituted version of $e'$, $P$ must also hold for $e$. □

While this is a somewhat involved proof, it is important to remember that the details of the proof are irrelevant to the practical use of our approach. The "black box" variables introduced can be treated as just that. We only provide a model for them to show soundness. Similarly, the only important thing about the code used at recursive calls is that it clearly decrements the recursive argument. The exact value of that argument need not be used in proofs, and indeed it *cannot* be used, because we do not provide any axioms about *depth*.

We can apply a further simplification by using a more literal translation on parts of the input program that are already "simple" enough. More specifically, where a program is a set of mutually recursive definitions, we can use a simpler translation for every definition with no transitive path of references to a variable defined with some imperative or non-primitive-recursive code. In fact, our experience suggests that almost all parts of a well-factored analysis will fall into this category, leaving just a few of the (generally most interesting) parts to translate with full generality.

## 4 Soundness Certification

We use the techniques described in the previous section to convert the ML data type `abs` to a description of the abstract domain $\mathcal{A}$ in the logic of the proof-assistant. Similarly, we convert the `ainv` value into a set $\mathcal{A}_I \subseteq \mathcal{A}$. Finally, we model the transition function `astep` as an abstract transition relation $\rightsquigarrow \subseteq \mathcal{A} \times 2^{\mathcal{A}}$ such that $a \rightsquigarrow A$ whenever $\texttt{astep}(a) = \texttt{Succ}\ A$. We will abuse notation slightly and identify sets and lists where convenient.

We prove soundness of the abstract transition relation with respect to a concrete transition relation. Let $(\mathcal{C}, \mathcal{C}_0, \mapsto)$ be a *transition system* for the concrete machine. In particular, $\mathcal{C}$ is a domain of states; $\mathcal{C}_0$ is the set of allowable initial states; and $\mapsto$ is a one-step transition relation. These elements are provided in the proof-assistant logic and are trusted. We build whatever safety policy interests us into $\mapsto$ in the usual way; we disallow transitions that would violate the policy, so that errors are modeled as "being stuck." This is the precise way in which one can specify the trusted *safety policy* for the certified program verifiers (Sect. 5).

To certify the soundness of the program analyzer, the certifier developer needs to provide additionally (in the form of a Coq definition) a *soundness relation* $\simeq \subseteq \mathcal{C} \times \mathcal{A}$ (written as $\sigma$ in [CC92]), such that $c \simeq a$ holds if the abstract state $a$ is a sound abstraction of the concrete state $c$. To demonstrate $\simeq$ is indeed sound, the author also provides proofs (in Coq) for the following standard, local soundness properties of abstract interpretations and bi-simulations.

*Property 1 (Initialization).* For every $c \in \mathcal{C}_0$, there exists $a \in \mathcal{A}_I$ such that $c \simeq a$.

The initialization property assures us that the abstract interpretation includes an appropriate invariant for every possible concrete initial state.

*Property 2 (Progress).* For every $c \in \mathcal{C}$ and $a \in \mathcal{A}$ such that $c \simeq a$, if there exists $A' \subseteq \mathcal{A}$ such that $a \rightsquigarrow A'$, then there exists $c' \in \mathcal{C}$ such that $c \mapsto c'$.

Progress guarantees that, whenever an abstract state is not stuck, any corresponding concrete states are also not stuck.

*Property 3 (Preservation).* For every $c \in \mathcal{C}$ and $a \in \mathcal{A}$ such that $c \simeq a$, if there exists $A' \subseteq \mathcal{A}$ such that $a \rightsquigarrow A'$, then for every $c' \in \mathcal{C}$ such that $c \mapsto c'$ there exists $a' \in (A' \cup \mathcal{A}_I)$ such that $c' \simeq a'$.

Preservation guarantees that, for every step made by the concrete machine, the resulting concrete state matches one of the successor states of the abstract machine. Preservation is only required when the abstract machine does not reject the program. This allows the abstract machine to reject some safe programs, if it so desires. It is important to notice that, in order to ensure termination, the `astep` function (and thus the $\rightsquigarrow$ relation) only returns those successor abstract states that are not already part of the initial abstract states `ainv`. To account for this aspect, we use $\mathcal{A}_I$ in the preservation theorem.

Together, these properties imply the global soundness of the certifier that implements this abstract interpretation [CC77], stated as following:

**Theorem 2 (Certification soundness).** *For any concrete state $c \in \mathcal{C}$ reachable from an initial state in $\mathcal{C}_0$, the concrete machine can make further progress. Also, if $c$ has the same program counter as a state $a \in \mathcal{A}_I$, then $c \simeq a$.*

### 4.1 Proof of Certification Soundness

In this subsection, we re-state the standard soundness theorem of abstract interpretation more thoroughly and sketch the proof that our conditions imply it.

**Theorem 3 (Fixpoint).** *If the* `top` *function shown in Fig. 3 completes successfully, then the set $A_F \in \mathcal{A}$ of abstract states that were processed has the following properties: $A_F$ contains at most one element for each value of the program counter, $\mathcal{A}_I \subseteq A_F$, and for each $a \in A_F$ there exists $A' \subseteq A_F$ such that $a \rightsquigarrow A'$ (i.e.,* `astep` *succeeds on $a$).*

This theorem be be proved easily by induction on the number of iterations of the `top` function. Now we can state the soundness theorem for certification.

**Theorem 4 (Soundness).** *If the Initialization, Progress, and Preservation properties hold, then any execution of the concrete machine starting in an initial state $\mathcal{C}_0$ will not get stuck, and at any point $c'$ in such an execution there exists $a' \in A_F$ such that $c' \simeq a'$.*

The proof of this theorem is by induction on the number of execution steps of the concrete machine. The Initialization property along with the fact that $\mathcal{A}_I \subseteq A_F$ (from the Fixpoint theorem) takes care of the base case. The induction

case follows from the Progress and Preservation lemmas and from the Fixpoint theorem. The fact that $A_I$ are indeed invariants of the program (Theorem 2) follows now using the fact that $A_F$ contains at most one invariant for each program counter.

### 4.2 Certifying the Example Java Bytecode Verifier

To give a better idea about what these obligations mean in practice, we sketch how the proof goes for our running example. Before doing so, we fill in some details about our concrete semantics, which the author of a verifier is not allowed to modify.

*Concrete Semantics.* Let's assume that the set of initial concrete states $\mathcal{C}_0$ is the set of all valid entry states to a special static `main`() method. The concrete transition relation $\mapsto$ is a standard one. Our safety policy will be memory safety, with some predetermined set of memory addresses the program is allowed to read or write. The only "stuck states" of $\mapsto$ will be those that try to jump to addresses that do not point to instructions of the program or try to read or write field addresses outside the set of valid addresses.

*Proof Sketch.* We now consider the proof that our example verifier is sound. First, we need to define $\simeq$. We assume a fixed object layout convention that includes a dynamic type tag for each object, all of which are allocated in the heap. We define $a \simeq c$ to hold if and only if:

1. the program counters of the two states agree;
2. the concrete state's register values and heap obey the object layout convention, as given by the `fieldOf` function;
3. every register of type `Class`($cls$) in $a$ contains in $c$ a valid heap pointer to an object whose dynamic type tag denotes a subclass of $cls$; and
4. every register of type `Cont`($a'$) in $a$ contains in $c$ a pointer to one of the *fixed* set of labels among those in `ainv` whose preconditions are compatible with abstract state $a'$.

Now we can prove the Initialization property for `ainv`. A reasonable definition of `ainv` in Fig. 5 would always generate an invariant for the entry point to the `main` method. From this, we see that the axiom about `main`'s presence allows us to conclude that $\mathcal{A}_I$ contains some $a$ that starts at `main` and has an expected, fixed precondition. Assuming that we defined $\simeq$ correctly, we can prove that this precondition includes every valid concrete entry state for `main`.

Regarding the proof of Progress and Preservation for `astep`, it's easiest to prove this by cases, mirroring the main `case` expression of `astep`. We sketch the approach by considering three representative cases.

First, consider the register-to-register move. We always accept such an instruction, and we model its effect by incrementing the program counter and copying type information for the source register to the slot for the destination

register. Let's call the abstract state on entry $a$ and the result state $a'$. Similarly, let $c$ be such that $a \simeq c$ and $c'$ such that $c \mapsto c'$, following the concrete semantics of this instruction. The Progress property comes for free here since this kind of instruction can't violate the safety policy. As for Preservation, it's easy to see that almost all of a proof of $a' \simeq c'$ follows from our assumption of $a \simeq c$, as the heap layout is preserved by this operation. We are sure to update the program counter correctly. The remaining parts of $\simeq$'s definition, regarding register values, are handled by copying of type information in the abstract world to mirror copying of value information in the concrete world.

Now consider the field read instruction. We update one register's type based on the type of the field we read, after checking that the object we read from has the expected type. We also increment the program counter, as in the last case. Here, Progress follows from the part of $\simeq$'s definition that tells us that a register with a `Class` type must point to a valid object of that class, as well as the part of that definition that assures us that the heap is well-laid-out. This also allows us to conclude that the field value we read has the proper type, which lets us prove Preservation similarly to how we did so in the last case.

Finally, let's look at the most interesting case, the indirect jump. We get Progress from our $\simeq$ invariant that a register with type $\texttt{Cont}(a')$ gets its value from a valid code label. Next, we must use the part of Preservation's definition that allows the case where $c \mapsto c'$ and $a'' \simeq c'$ for some $a'' \in \mathcal{A}_I$. To prove Progress, we make use of two facts:

- We know from our $a \simeq c$ assumption that we are jumping to the address of some label $\ell$ whose precondition $a''$ is compatible with $a'$, since we have verified that the target register has type $\texttt{Cont}(a')$.
- We know that *every label has a corresponding state in* $\mathcal{A}_I$, so in particular, $\ell$ must correspond to some $a'' \in \mathcal{A}_I$.

This assures us that there is $a'' \in \mathcal{A}_I$ such that $a$ is compatible with $a'$ (thanks to the explicit `subAbs` check) and $a'$ is compatible with $a''$. Transitively, we have that the result abstract state is compatible with $a''$.

## 5 Applications to Mobile-Code Safety

Language-based security mechanisms have gained acceptance for enforcing basic but essential safety properties, such as memory and type safety, for untrusted mobile code. The most widely deployed solution for mobile code safety is bytecode verification, as in the Java Virtual Machine (JVM) [LY97] or the Microsoft Common Intermediate Language (MS-CIL) [GS01]. A bytecode verifier uses a form of abstract interpretation to track the types of machine registers, and to enforce memory and type safety. The main limitation of this approach is that we must trust the soundness of the bytecode verifier. In turn, this means that we cannot easily change the verifier and its enforcement mechanism. This effectively forces the clients of a code receiver to use a fixed type system and often even a fixed source language for mobile code. Programs written in other source languages

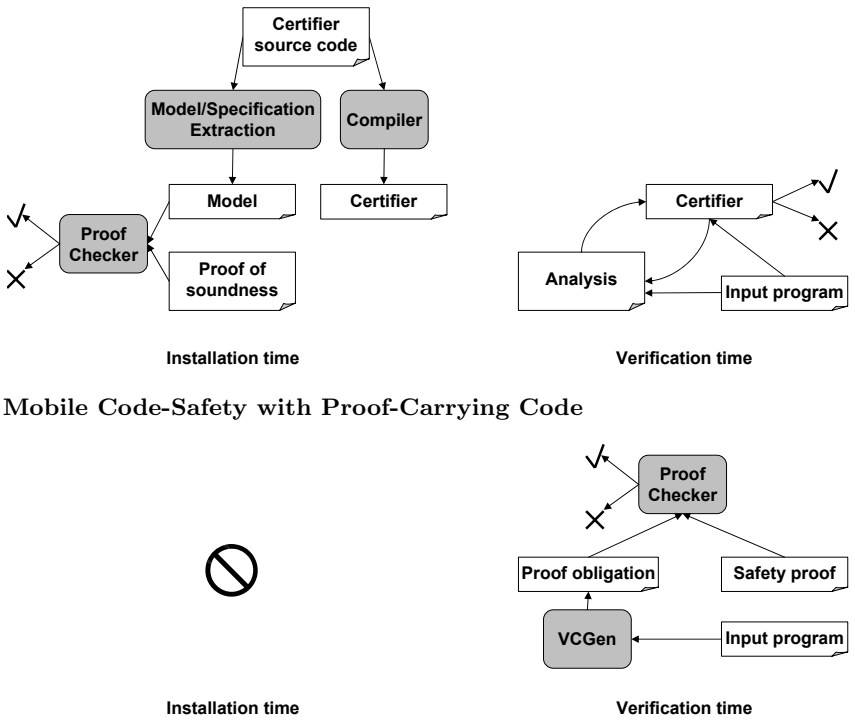


**Fig. 13.** Comparing and contrasting our certified verifier architecture (Fig. 1) and traditional proof-carrying code implementations. The trusted code base is shown shaded

can be compiled into the trusted intermediate language but often in unnatural ways with a loss of expressiveness and performance [BKR99,GC00,Bot98].

A good example is the MS-CIL language, which is expressive enough to be the target of compilers for C#, C and C++. Compilers for C# produce intermediate code that can be verified, while compilers for C and C++ use intermediate language instructions that are always rejected by the built-in bytecode verifier. In this latter case, the code may be accepted if the producer of the code can provide an explicit proof that the code obeys the required safety policy and the code receiver uses proof-carrying code [App01,HST+02,Nec97].

Existing work on proof-carrying code (PCC) attests to its versatility, but often fails to address the essential issue of how the proof objects are obtained. In the Touchstone system [CLN+00], proofs are generated by a special theorem prover with detailed knowledge about Java object layout and compilation strategies. The Foundational PCC work [App01,HST+02] eliminates the need to hard-code and trust all such knowledge, but does so at the cost of increasing many times the proof generation burden. Both these systems also incur the cost of transmitting proofs. The Open Verifier project [CCNS05] proposes to send

with the code not per-program proofs but proof generators to be run at the code receiver end for each incoming program. The generated proofs are then checked by a trusted proof checker, as in a standard PCC setup.

Using certified program analyses we can further improve this process. The producer of the mobile code writes a safety-policy verifier customized for the exact compilation strategy and safety reasoning used in the generation of the mobile code. This verifier can be written in the form of a certified program analysis, whose abstract transition fails whenever it cannot verify the safety of an instruction. For example, we discuss in Sect. 6 cases when the program analysis is a typed assembly language checker, a bytecode verifier, or an actual PCC verification engine relying on annotations accompanying the mobile code.

The key element is the soundness proof that accompanies an analysis, which can be checked automatically. At verification time, the now-trusted program analyzer is used to validate the code, with no need to manipulate explicit proof objects. This simplifies the writing of the validator (as compared with the proof-generating theorem prover of Touchstone, or the Open Verifier). Fig. 13 highlights the main differences between our certified verifier architecture and traditional PCC implementations. We also show in Sect. 6 that this reduces the validation time by more than an order of magnitude.

We point out here that the soundness proof is with respect to the trusted concrete semantics. By adding additional safety checks in the concrete semantics (for instance, the logical equivalents of dynamic checks that would enforce a desired safety policy), the code receiver can construct customized safety policies.

## 6 Case Studies

In this section, we present case studies of applying certified program analyzers to mobile code security. We describe experience with verifiers for typed assembly language, Java bytecode, and proof-carrying code.

We have developed a prototype implementation of the certified program analysis infrastructure. The concrete language to be analyzed is the Intel x86 assembly language. The specification extractor is built on top of the front-end of the OCaml compiler, and it supports a large fragment of the ML language. The most notable features not supported are the object-oriented features. In addition to the 3000-line extractor, the trusted computing base includes the whole OCaml compiler and the Coq proof checker, neither of which is designed to be foundationally small. However, our focus here has been on exploring the ease of use and run-time efficiency of our approach. We leave minimizing the trusted base for future work.

*Typed Assembly Language.* Our first realistic use of this framework involved Typed Assembly Language. In particular, we developed and proved correct a verifier for TALx86, as provided in the first release of the TALC tools from Cornell [MCG+03]. This TAL includes several interesting features, including continuation, universal, existential, recursive, product, sum, stack, and array

types. Our implementation handles all of the features used by the test cases distributed with TALC, with the exception of the modularity features, which we handle by "hand-linking" multiple-file tests into single files. TALC includes compilers to an x86 TAL from Popcorn (a safe C dialect) and mini-Scheme. We used these compilers unchanged in our case study.

We implemented a TALx86 verifier in 1500 lines of ML code. This compares favorably with the code size of the TALC type checker, which is about 6000 lines of OCaml. One of us developed our verifier over the course of two months, while simultaneously implementing the certification infrastructure. We expect that it should be possible to construct new verifiers of comparable complexity in a week's time now that the infrastructure is stable.

We also proved the local soundness properties of this implementation in 15,000 lines of Coq definitions and proof scripts. This took about a month, again interleaved with developing the trusted parts of the infrastructure. We re-used some definitions from a previous TAL formalization [CCNS05], but we didn't re-use any proofs. It's likely that we can significantly reduce the effort required for such proofs by constructing some custom proof tactics based on our experiences. We don't believe our formalization to be novel in any fundamental way. It uses ideas from previous work on foundational TAL [AF00,HST+02,Cra03]. The main difference is that we prove the same basic theorems about the behavior of an implementation of the type checker, instead of about the properties of inference rules. This makes the proofs slightly more cumbersome, but, as we will see, it brings significant performance improvement. As might be expected, we found and fixed many bugs in the verifier in the course of proving its soundness. This suggests that our infrastructure might be useful even if the developer is only interested in debugging his analysis.

| | Conv | CPV | PCC |
|---|---|---|---|
| Up to 200 (13) | 0 | 0.01 | 0.07 |
| 201-999 (7) | 0.01 | 0.02 | 0.24 |
| 1000 and up (6) | 0.04 | 0.08 | 1.73 |

**Table 1.** Average verifier running times (in seconds)

Table 1 presents some verification-time performance results for our implementation, as average running times for inputs with particular counts of assembly instructions. We ran a number of verifiers on the test cases provided with TALC, which used up to about 9000 assembly instructions. First, the type checker included with TALC finishes within the resolution of our timing technique for all cases, so we don't include results for it. While this type checker operates on a special typed assembly language, the results we give are all for verifying native assembly programs, with types and macro-instructions used as meta-data. As a result, we can expect that there should be some inherent slow-down, since some

TAL instructions must be compiled to multiple real instructions. The experiments were performed on an Athlon XP 3000+ with 1 GB of RAM, and times are given in seconds. We give times for "Conventional (Conv)," a thin wrapper around the TALC type checker to make it work on native assembly code; "CPV," our certified program verifier implementation; and "PCC," our TALx86 verifier implementation from previous work [CCNS05], in which explicit proof objects are checked during verification.

The results show that our CPV verifier performs comparably with the conventional verifier, for which no formal correctness proof exists. It appears our CPV verifier is within a small constant factor of the conventional verifier. This constant is likely because we use an inefficient, Lisp-like serialization format for including meta-data in the current implementation. We expect this would be replaced by a much faster binary-encoded system in a more elaborate version.

We can also see that the certified verifier performs much better than the PCC version. The difference in performance is due to the cost required to manipulate and check explicit proof objects during verification. To provide evidence that we aren't comparing against a poorly-constructed straw man, we can look to other FPCC projects. Wu, Appel, and Stump [WAS03] give some performance results for their Prolog-based implementation of trustworthy verifiers. They only present results on input programs of up to 2000 instructions, with a running time of .206 seconds on a 2.2 GHz Pentium IV. This seems on par with our own PCC implementation. While their trusted code base is much smaller than ours, since we require trust in our specification extractor, there is hope that we can achieve a similarly small checking kernel by using techniques related to certifying compilation.

*Java Bytecode Verification.* We have also used our framework to implement a partial Java Bytecode Verifier (JBV) in about 600 lines of ML. It checks most of the properties that full JBVs check, mainly excluding exceptions, object initialization, and subroutines. Our implementation's structure follows closely that of our running example from Sect. 2. Its `ainv` begins by calling an OCaml function that calculates a fixed point using standard techniques. Like in our example, the precise code here doesn't matter, as the purpose of the function is to populate a hash table of function preconditions and control-flow join point invariants. With this information, our `astep` function implements the standard typing rules for JBVs.

While we have extracted complete proof obligations for the implementation, we have only begun the process of proving them. However, to make sure we are on track to an acceptable final product, we have performed some simple benchmarks against the bytecode verifier included with Blackdown Java for Linux. We downloaded a few Java-only projects from SourceForge and ran each verifier on every class in each project.

On the largest that our prototype implementation could handle, MegaMek, our verifier finishes in 5.5 seconds for checking 668,000 bytecode instructions, compared to 1 second for the traditional verifier. First, we note that both times are relatively small in an absolute sense. It probably takes a user considerably

```
fun checkProof (prf: proof) (p: pred) : bool = ...
fun astep (a: abs) : result =
 case instrAt a.pc of
   RegReg(r1, r2) => Succ [{
     pc = a.pc + 1;
     a = And(Eq(r1,r2),Exists(x,[x/r1]a.a)) }]
 | Jump l =>
     let dest = getInvar l in
     let prf = fetchProof l in
     if checkProof (prf, Imply(a.a, dest)) then
       Succ [ ]
     else Fail
```

**Fig. 14.** A fragment of a certifier for PCC

longer to download a software package than to verify it with either method. We also see that our verifier is only a small factor away from matching the traditional approach, whose performance we know empirically that users seem willing to accept. No doubt further engineering effort could close this gap or come close to doing so.

*Proof-Carrying Code.* We can even implement a version of Foundational PCC in our framework: for each basic block the mobile code contains an invariant for the start of the block, and a proof that the strongest postcondition of the start invariant along the block implies the invariant for the successor block. The abstract state `abs` of the certifier consists of a predicate written in a suitable logic, intended to be the strongest postcondition at the given program point. The `ainv` is obtained by reading invariants from a data segment accompanying the mobile code.

Fig. 14 shows a fragment of the code for `astep`, which calculates the strongest postcondition for every instruction. At a jump we fetch the invariant for the destination, a proof, and then check the proof. To prove soundness, we only need to ensure that `getInvar` returns one of the invariants that are part of `ainv`, and that the `checkProof` function is sound. More precisely, whenever the call to `checkProof` returns true, then any concrete state that satisfies `a.a` also satisfies `dest`. In particular, we do not care at all how `fetchProof` works, where it gets the proof from, whether it decrypts or decompresses it first, or whether it actually produces the proof itself. This soundness proof for `checkProof` is possible and even reasonably straightforward, since we are writing our meta-proofs in Coq's more expressive logic.

## 7 Related Work

*Toward Certified Program Analyses.* The Rhodium system developed by Lerner *et al.* [LMRC05] is the most similar with respect to the overall goal of our work—that of providing a realistic framework for certified program analyses.

However, they focus on simpler compiler analysis problems whose soundness can be proved by today's automated methods. We expect that our proofs can similarly be automated when our framework is used for the kinds of analyses expressible in Rhodium-style domain specific languages.

Several systems have been developed for specifying program analyses in domain-specific languages and generating code from these specifications [Las03]. Again, the expressiveness of these systems is very limited compared to what is needed for standard mobile code safety problems.

In the other direction, we have the well-established body of work dealing with extracting formal verification conditions from programs annotated with specifications. Especially relevant are the Why [Fil03] and Caduceus [FM04] tools, which produce Coq proof obligations as output.

There has been a good amount of work on constructing trustworthy verifiers by extracting their code from constructive proofs of soundness. Cachera *et al.* [CJPR04] extracted a data-flow analysis from a proof based on a general constraint framework. Klein and Nipkow [KN01] and Bertot [Ber01] have built certified Java bytecode verifiers through program extraction/code generation from programs and proofs in Isabelle and Coq, respectively. None of these publications present any performance figures to suggest that their extracted verifiers scale to real input sizes

*Enforcing Mobile-Code Safety.* As alluded to earlier, most prior work in Foundational Proof-Carrying Code has focused on the generality and expressivity of various formalisms, including the original FPCC project [AF00], Syntactic FPCC [HST+02], and Foundational TALT [Cra03]. These projects have given convincing arguments for their expressiveness, but they have not yet demonstrated a scalable implementation. Some recent research has looked into efficiency considerations in FPCC implementations, including work by Wu, Appel, and Stump [WAS03] and our own work on the Open Verifier [CCNS05].

The architecture proposed by Wu, Appel, and Stump is fairly similar to the architecture we propose, with the restriction that verifiers must be implemented in Prolog. In essence, while we build in an abstract interpretation engine, Wu *et al.* build in a Prolog interpreter. We feel that it is important to support verifiers developed in more traditional programming languages. Also, the performance figures provided by Wu *et al.* have not yet demonstrated scalability.

Our past work on the Open Verifier has heavily influenced the design of the certified program analysis architecture. Both approaches build an abstract interpretation engine into the trusted base and allow the uploading of customized verifiers. However, the Open Verifier essentially adheres to a standard PCC architecture in that it still involves proof generation and checking for each mobile program to be verified, and it pays the usual performance price for doing this.

## 8 Conclusion

We have presented a strategy for simplifying the task of proving soundness not just of program analysis algorithms, but also of their implementations. We be-

lieve that starting with the implementation and extracting natural proof obligations will allow developers to fine tune non-functional aspects of the code, such as performance or debugging instrumentation.

Certified program analyses have immediate applications for developing certified program verifiers, such that even untrusted parties can customize the verification process for untrusted code. We have created a prototype implementation and used it to demonstrate that the same infrastructure can support in a very natural way proof-carrying code, type checking, or data-flow based verification in the style of bytecode verifiers. Among these, we have completed the soundness proof of a verifier for x86 Typed Assembly Language. The performance of our certified verifier is quite on par with that of a traditional, uncertified TALx86 type checker. We believe our results here provide the first published evidence that a foundational code certification system can scale.

## References


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

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proc. of the 16th Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[BCDdS02] G. Barthe, P. Courtieu, G. Dufay, and S. de Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proc. of the 9th International Conference on Algebraic Methodology and Software Technology*, September 2002.

[Ber01] Yves Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proc. of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 14–24, July 2001.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proc. of the International Conference on Functional Programming*, pages 129–140, June 1999.

[Bot98] Per Bothner. Kawa — compiling dynamic languages to the Java VM. In *Proc. of the FreeNIX Track: USENIX 1998 annual technical conference*, 1998.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symposium on Principles of Programming Languages*, pages 234–252, January 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CCN06] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Proc. of the 7th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2006.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. of the 2nd Workshop on Types in Language Design and Implementation*, January 2005.

[CJPR04] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In David A. Schmidt, editor, *Proc. of the 13th European Symposium on Programming*, volume 2986 of *LNCS*, pages 385–400, March 2004.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 95–107, May 2000.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proc. of the 30th Symposium on Principles of Programming Languages*, pages 198–212, January 2003.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Proc. of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 15–29, November 2004.

[GC00] K. John Gough and Diane Corney. Evaluating the Java virtual machine as a target for languages other than Java. In *Joint Modula Languages Conference*, September 2000.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proc. of the 28th Symposium on Principles of Programming Languages*, pages 248–260, January 2001.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proc. of the 17th Symposium on Logic in Computer Science*, pages 89–100, July 2002.

[KN01] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency – practice and experience*, 13(1), 2001.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298(3):583–626, 2003.

[Las03] John H. E. F. Lasseter. Toolkits for the automatic construction of data flow analyzers. Technical Report CIS-TR-04-03, University of Oregon, 2003.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd Symposium on Principles of Programming Languages*, pages 364–377, 2005.

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

[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.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. of the 24th Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[NJM+02] George C. Necula, Ranjit Jhala, Rupak Majumdar, Thomas A. Henzinger, and Westley Weimer. Temporal-safety proofs for systems code. In *Proc. of the Conference on Computer Aided Verification*, November 2002.

[Pau94] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[Ros03] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

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