Automated Techniques for Provably Safe Mobile Code[⋆]

Christopher Colby

Fast Forward Networks, 75 Hawthorne Lane, Suite 601, San Francisco, CA 94105

Karl Crary Robert Harper Peter Lee Frank Pfenning[*]

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891

Abstract

We present a general framework for provably safe mobile code. It relies on a formal definition of a safety policy and explicit evidence for compliance with this policy which is attached to a binary. Concrete realizations of this framework are proof-carrying code (PCC), where the evidence for safety is a formal proof generated by a certifying compiler, and typed assembly language (TAL), where the evidence for safety is given via type annotations propagated throughout the compilation process in typed intermediate languages. Validity of the evidence is established via a small trusted type checker, either directly on the binary or indirectly on proof representations in a logical framework (LF).

Key words: type systems, type safety, certifying compilers, typed assembly language, proof-carrying code

[⋆] This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
[*] Corresponding author.
Email addresses: colby@ffnet.com (Christopher Colby), crary@cs.cmu.edu (Karl Crary), rwh@cs.cmu.edu (Robert Harper), petel@cs.cmu.edu (Peter Lee), fp@cs.cmu.edu (Frank Pfenning).

1 Introduction

Integrating software components to form a reliable system is a long-standing fundamental problem in computer science. The problem manifests itself in numerous guises:

(1) How can we dynamically add services to an operating system without compromising its integrity?
(2) How can we exploit existing software components when building a new application?
(3) How can we support the safe exchange of programs in an untrusted environment?
(4) How can we replace components in a running system without disrupting its operation?

These are all questions of *modularity*. We wish to treat software components as "black boxes" that can be safely integrated into a larger system without fear that their use will compromise, maliciously or otherwise, the integrity of the composite system. Put in other terms, we wish to ensure that the behavior of a system remains predictable even after the addition of new components.

Three main techniques have been proposed to solve the problem of safe component integration:

(1) **Run-time checking**. Untrusted components are monitored at execution time to ensure that their interactions with other components are strictly limited. Typical techniques include isolation in separate hardware address spaces and software fault isolation [1]. These methods impose serious performance penalties in the interest of safety. Moreover, there is often a large semantic gap between the low-level properties that are guaranteed by checking (*e.g.*, address space isolation) and the high-level properties that are required (*e.g.*, black box abstraction).
(2) **Source-language enforcement**. All components are required to be written in a designated language that is known, or assumed, to ensure "black box" abstraction. These techniques suffer from the requirement that all components be written in a designated, safe language, a restriction that is the more onerous for lack of widely-used safe languages. Moreover, one must assume not only that the language is properly defined, but also that its implementation is correct, which, in practice, is never the case.
(3) **Personal authority**. No attempt is made to enforce safety, rather the component is underwritten by a person or company willing to underwrite its safety. Digital signature schemes may be used to authenticate the underwritten code. In practice few, if any, entities are willing to make

assurances for the correctness of their code.

What has been missing until now is a careful analysis of *what* is meant by safe code exchange, rather than yet another proposal for *how* one might achieve a vaguely-defined notion of safe integration. Our contention is that safe component integration is fundamentally a matter of *proof*. To integrate a component into a larger system, the code recipient wishes to know that the component is suitably well-behaved — that is, compliant with a specified *safety policy*. In other words, it must be apparent that the component satisfies a *safety specification* that governs its run-time behavior. Checking compliance with such a safety specification is a form of *program verification* in which we seek to prove that the program complies with the given safety policy.

When viewed as a matter of verification, the question arises as to who (the code producer or the code recipient) should be responsible for checking compliance with the safety policy. *The problem with familiar methods is that they impose the burden on the recipient.* The code producer insists that the recipient employ run-time checks, or comply with the producer's linguistic restrictions, or simply trust the producer to do the right thing. But, we argue, this is exactly the wrong way around. To maximize flexibility we wish to exploit components from many different sources; it is unreasonable to expect that a code recipient be willing to comply with the strictures of each of many disparate methods. Rather, we argue, *it is the responsibility of the code producer to demonstrate safety.* It is (presumably) in the producer's interest for the recipient to use its code. Moreover, it is the producer's responsibility (current practices notwithstanding) to underwrite the safety of its product. In our framework we shift the burden of proof from the recipient to the producer.

Having imposed the burden of proof on the producer, how is the consumer to know that the required obligations have been fulfilled? One method is to rely on trust — the producer signs the binary, affirming the safety of the component. This suffers from the obvious weakness that the recipient must trust not only the producer's integrity, but also must trust the tools that the producer used to verify the safety of the component. Even with the best intentions, it is unlikely that the methods are foolproof. Consequently, few producers are likely to make such a warrant, and few consumers are likely to rely on the code they receive. A much better method is one that we propose here: require the producer to provide a formal representation of the *proof* that the code is compliant with the safety policy. After all, if the producer did carry out such a proof, it can easily supply the proof to the consumer. Moreover, the recipient can use its own tools to check the validity of the proof to ensure that it really is a genuine proof that the given code complies with the safety specification. Importantly, *it is much easier to check a proof than it is to find a proof.* Therefore the code recipient need only trust its *own* proof checker, which is, if the method is to be effective, much simpler than the tools required to find the proof in the first place.

The message of this paper is that this approach can, in fact, be made to work in practice. We are exploring two related techniques for implementing our approach to safe component exchange: *proof-carrying code* and *typed object code*. In both cases mobile code is annotated with a formal warrant of its safety, which can be easily checked by the code recipient. To produce such a warrant, we are exploring the construction of *certifying compilers* that produce suitably-annotated object code. Such a compiler could be used by a code producer to generate certified object code. Two points should be kept in mind when reading this paper:

(1) The tools and techniques of logic, type theory, and semantics are indispensable.
(2) These methods have been implemented and are available today.

2 Safety Infrastructures

The first component in a system for safe mobile code is the *safety infrastructure*. The safety infrastructure is the piece of the system that actually ensures the safety of mobile code before execution. It forms the *trusted computing base* of the system, meaning that all consumers of mobile code install it and depend on it, and therefore it must work properly. Any defect in the trusted computing base opens a possible security hole in the system.

A fundamental concern in the design of the trusted computing base is that it be small and simple. Large and/or complicated code bases are very likely to contain bugs, and those bugs are likely to result in exploitable security holes. For us to have confidence in our safety infrastructure, its trusted components must be small and simple enough that they are likely to be correct.

The design of the safety infrastructure consists of three parts. First, one must define a safety policy. Second, one specifies what will be acceptable as evidence of compliance with the safety policy. Suppliers of mobile code will then be required also to supply evidence of compliance in an acceptable form. Third, one must build software that is capable of automatically checking whether purported evidence of safety is actually valid.

2.1 Safety Policies

The first task in the design of the safety infrastructure is to decide what properties mobile code must satisfy to be considered safe. In this paper we will consider a relatively simple safety policy, consisting of *memory safety, control-flow safety,* and *type safety.*

(1) Memory safety is the property that a program never dereferences an invalid pointer, never performs an unaligned memory access, and never reads or writes any memory locations to which it has not been granted access. This property ensures the integrity of all data not available to the program, and also ensures that the program does not crash due to memory accesses.
(2) Control-flow safety is the property that a program never jumps to an address not containing valid code, and never jumps to any code to which it has not been granted access. This property ensures that the program does not jump to any code to which it is not allowed (*e.g.,* low-level system calls), and also ensures that the program does not crash due to jumps.
(3) Type safety is the property that every operation the program performs is performed on values of the appropriate type. Strictly speaking, this property subsumes memory and control-flow safety (since memory accesses and jumps are program operations), but it also makes additional guarantees. For example, it ensures that all (allowable) system calls are made using appropriate values, thereby ruling out attacks such as buffer overruns on other code in the system. The additional guarantees provided by type safety are often very expensive to obtain using dynamic means, but the static means we discuss in this paper can provide them at no additional cost.

Stronger safety policies are also possible, including guarantees of the integrity of data stored on the stack [2], limits on resource consumption [3,4], and policies specified by allowable traces of program operations [5]. However, for policies such as these, the evidence of compliance (which we discuss in the next section) can be more complicated, thereby requiring greater expense both to produce and to verify that evidence, and possibly reducing confidence in the system's correctness. Thus the choice of safety policy in a practical system involves important trade-offs.

It is also worth observing that stronger policies are not always better if they rule out too many programs. For example, a policy that rejects *all* programs provides great safety (and is cheap to implement), but is entirely useless for a safety infrastructure. Therefore, it is important to design safety policies to allow as many programs as possible, while still providing sufficient safety.

2.2 Evidence of Compliance

The safety policy establishes what properties mobile programs must satisfy in order to be permitted to execute on a host. However, it is impossible in general for a code consumer to determine whether an arbitrary program complies with that policy. Therefore, we require that suppliers of mobile code assist the consumer by providing evidence that their code complies with the safety policy. This evidence, which we may think of as a *certificate of safety,* is packaged together with the mobile program and the two together are referred to as *certified code.* Upon obtaining certified code, the code consumer (automatically) verifies the validity of the evidence before executing the program code.

The second task in the design of the safety infrastructure is to decide what form the evidence of compliance must take. This decision is made in the light of several considerations:

(1) Since evidence of safety must be transferred over the network along with the program code they certify, we wish the evidence to be as small as possible in order to minimize communication overhead.
(2) Since evidence must be checked before running any program code, we desire verification of evidence to be as fast as possible. Clearly, smaller evidence can lead to faster checking, but we can also speed evidence verification by careful design of the form of evidence.
(3) As discussed above, the evidence verifier is an essential part of the trusted computing base; it must work properly or there will be a potential security hole in the system. For us to have confidence that the verifier works properly, it must be simple, which means that the structure of the evidence it checks must also be simple. Thus, not only is simplicity desirable from an aesthetic point of view, but it is also essential for the system to work.
(4) Finally, to have complete confidence that our system provides the desired safety, we must prove with mathematical rigor that programs carrying acceptable evidence of safety really do comply with the safety policy. This proof is at the heart of the safety guarantees that the system provides. For such proofs to be feasible, the structure of the evidence must be built on mathematical foundations.

In light of these considerations, we now discuss two different forms that evidence of compliance may take: explicit proofs, which are employed in the Proof-Carrying Code infrastructure [6], and type annotations, which are employed in the Typed Assembly Language infrastructure [7,2].

Explicit Proofs

The most direct way to provide evidence of safety is to provide an explicit formal proof that the program in question complies with the safety policy. This is the strategy employed by Proof-Carrying Code (PCC). It requires a formal language in which safety proofs can be expressed. Any such language should be designed according to the following criteria.

Effective Decidability: It should be efficiently decidable if a given object represents a valid safety proof.

Compactness: Proofs should have small encodings.

Generality: The representation language should permit proofs of different safety properties. Ideally, it should be open-ended so that new safety policies can be developed without a change in the trusted computing base.

Simplicity: The proof representation language should be as simple as possible, since we must trust its mathematical properties and the implementation of the proof checker.

Our approach has been to use the LF logical framework [8] to satisfy these requirements. A *logical framework* is a general meta-language for the representation of logical inferences rules and deductions. Various logics or theories can be specified in LF at a very high level of abstraction, simply by stating valid axioms and rules of inference. This provides *generality*, since we can separate the theories required for reasoning about safety properties such as arithmetic modulo 2^{32} or memory update and access from the underlying mechanism of checking proofs. It is also *simple*, since it is based on a pure, dependently typed λ-calculus whose properties have been deeply investigated [9,10].

Proofs in a logic designed for reasoning about safety properties are represented as terms in LF. Checking that a proof is valid is reduced to checking that its representation in the logical framework is well-typed. This can be carried out effectively even for very large proof objects. Experiments in certifying compilation [11] and decision procedures [12] yield proofs whose representation is more than 1 MB, yet can still be checked. On the other hand, proofs in LF are not compact without additional techniques for redundancy elimination. Following some general techniques [13], Necula [14] has developed optimized representations for a fragment of LF called LF_i which is sufficient for its use in PCC applications. The experimental results obtained so far have validated the practicality of this proof compression technique [11] for the safety policies discussed here. Current research [15] is aimed at extending and improving these methods to obtain further compression without compromising the simplicity of the trusted computing base.

Type Annotations

A second way to provide evidence of safety is using type annotations. In this approach a typing discipline is imposed on mobile programs, and the architects of the system prove a theorem stating that any program satisfying that type discipline will necessarily satisfy the safety policy as well [7]. However, determining whether a program satisfies a type discipline involves finding a consistent type scheme for the values in the program, and such a type scheme cannot be inferred in general. Therefore, in this approach programs are required to include enough type annotations for the type checker to reconstruct a consistent type scheme. Such type annotations constitute the evidence of safety, provided they are taken in conjunction with a theorem stating that well-typed programs comply with the safety policy.

A principal advantage of the type annotation approach over the explicit proof approach is that the soundness of the type system can be established once and for all. In contrast, validity of explicit proofs does not establish the soundness of the system of proof rules, and in practice the proof rules are freely customized to account for the safety requirements of each application. The main drawback of type annotations is that any program that violates a type system's invariants will not be typeable under that type system, and therefore cannot be accepted by the safety infrastructure, even if it is actually safe. With explicit proofs, such invariants are not built in, so it is possible to work around cases in which they do not hold.

The idea of using types to guarantee safety is by no means new. Many modern high-level languages (*e.g.,* ML, Modula-3, Java) rely on a type system to ensure that all legal programs are safe. Such languages have even been used for safety infrastructures; for example, the SPIN operating system [16] required that operating system extensions be written in Modula-3, thereby ensuring their safety. The drawback to using a high-level language to ensure safety is that programs are checked for safety before compilation, rather than after, thereby requiring that the entire compiler be included in the trusted computing base. As discussed above, the confidence that one can have in the safety architecture is inversely related to the size of its trusted computing base.

The Typed Assembly Language (TAL) infrastructure resolves this problem by employing a type safe *low-level* language. In TAL, a type discipline is imposed on executable code, and therefore the program code being checked for safety is the exact code that will be executed. There is no need to trust a compiler, because if the compiler is faulty and generates unsafe executables, those executables will be rejected by the type checker.

The principal exercise in developing a type system for executable code is to isolate low-level abstractions satisfying two conditions:

- The abstractions should be independently type checkable; that is, to whatever extent type checking of the abstractions depends on surrounding code and data, it should only depend on the *types* of that code and data, and not on additional information not reflected in the types.
- The atomic operations on the abstractions should be single machine instructions.

As an example consider function calls. High-level languages usually provide a built-in notion of functions. Functions can certainly be type checked independently, but they are not dealt with by a single machine instruction. Rather, function calls are processed using separate call and return instructions and the intervening code is by no means atomic: the return address is stored in accessible storage and can be modified or even disregarded. To satisfy the second condition, TAL's corresponding abstraction is the *code block,* and code blocks are invoked using a simple jump instruction. Functions are then composed from code blocks by writing code blocks with an explicit extra input containing the return address. (This decomposition corresponds to the well-known practice in high-level language of programming in continuation-passing style [17].) The first condition is satisfied by requiring code blocks to specify the types of their inputs, just as functions in high-level languages specify the types of their arguments and results. Without such specifications, it would be impossible to check the safety of a jump without inspecting the body of the jump's target.

For example, consider the TAL code below for computing factorial. This code does not exhibit many of the complexities of the TAL type system, but it serves to give the flavor of TAL programs. (More exhaustive examples appear in Morrisett, *et al.* [7,2].)

```
fact:
  code{r1:int,r2:{r1:int}}.
    mov r3,1          % set up accumulator for loop
    jmp loop
loop:
  code{r1:int,r2:{r1:int},r3:int}.
    bz r1,done        % check if done, branch if zero
    mul r3,r3,r1
    sub r1,r1,1
    jmp loop
done:
  code{r1:int,r2:{r1:int},r3:int}.
    mov r1,r3         % move accumulator to result register
    jmp r2            % return to caller
```

In this code, the type for a code block is written $\{\texttt{r1}:\tau_1,\ldots,\texttt{r}n:\tau_n\}$, indicating that the code may be called when registers `r1`, ..., `r`n contain values having type $\tau_1, \ldots, \tau_n$. The `fact` code block is given type {`r1`:*int*, `r2`:{`r1`:*int*}}, indicating that when `fact` is called, it must be given in `r1` an integer (the argument), and in `r2` a code block (the return address) that when called must be given an integer (the return value) in `r1`. When called, `fact` sets up an accumulator register (with type *int*) in `r3` and jumps to `loop`. Then `loop` computes the factorial and, when finished, branches to `done`, which moves the accumulator to the return address register (`r1`) and returns to the caller. In the final return to the caller, the extra registers `r2` and `r3` are forgotten to match the precondition on `r2`, which only mentions `r1`.

As an alternative to typed assembly language, one can also strike a compromise between high- and low-level languages by exploiting *typed intermediate languages* for safety [18]. Using typed intermediate languages enlarges the trusted computing base, since some part of the compiler must be trusted, but it loosens the second condition on type systems for executable code. This provides a spectrum of possible designs, the closer an intermediate language is to satisfying the second condition, the lesser the amount of the compiler that needs to be trusted. Moreover, as we discuss in Section 3, typed intermediate languages are valuable for automated certification, even if the end result is a typed executable.

2.3 Automated Verification

Since it plays such a central role for provably safe mobile code, we now elaborate on the mechanisms for verifying safety certificates.

Explicit Proofs

As discussed above, the Proof-Carrying Code infrastructure employs the LF logical framework. In the terminology of logical frameworks, a *judgment* is an object of knowledge which may be evident by virtue of a *proof*. Typical safety properties require only a few judgments, such as the truth of a proposition in predicate logic, or the equality of two integers.

In LF, a *judgment* of an object logic is represented by a *type* in the logical framework, and a *proof* by a *term*. If we have a proof P for a judgment J, then the representation $\overline{P}$ has type $\overline{J}$, where we write $\overline{()}$ for the representation function. The adequacy theorem for a representation function guarantees this property and its inverse: whenever we have a term M of type $\overline{J}$ then there is a proof P for J. Both directions are critical, because together they mean that we can reduce the problem of checking the validity of a proof P to verifying that its representation $\overline{P}$ is well-typed.

So in PCC, checking compliance with a safety policy can be reduced to type checking the representation of a safety proof in the logical framework.

But how does this technique allow for different safety policies? Since proofs are represented as terms in LF, an *inference rule* is represented as a function from the proofs of its premises to the proof of its conclusion. To represent a complete logical system we only need to introduce one type constant for each basic judgment and one term constant for each inference rule. The collection of these constant declarations is called a *signature.*[1] So a particular safety policy consists of a verification condition generator, which extracts a proof obligation from a binary, and signature in LF, which expresses the valid proof principles for the verification condition. This means that different policies can be expressed by different signatures, and that the basic engine that verifies evidence (the LF type checker) does not change for different policies. However, we do have to trust the correctness of the LF signature representing a policy—an inconsistent signature, for example, would allow arbitrary code to pass the safety check.

Type checking in LF is syntax-directed and therefore in practice quite efficient [13], especially if we avoid checking some information which can be statically shown to be redundant [14]. Currently, the Touchstone compiler for PCC discussed in Section 3.1 uses a small, efficient type checker for LF terms written in C. Related projects on proof-carrying code [19,20] and certifying decision procedures [12] use the Twelf implementation [21–23]. For more information on logical frameworks, see [24].

Type Annotations

In case the safety policy is expressed in the form of typing rules, checking compliance immediately reduces to type checking. In this case we have to carefully design the language of annotations so that type checking is practical. Generally, the more complicated the safety property the more annotations are required. Once the safety property is fixed, there is a trade-off between space and time: the more type annotations we have, the easier the type checking problem. One extreme consists of no type annotations at all, which means that type checking is undecidable. The other extreme is a full typing derivation (represented, for example, in the logical framework) which is quite similar to a proof in the PCC approach. For the safety policies we have considered so far, it has not been difficult to find appropriate compromises between these extremes that are both compact and permit fast type checking [25].

It is worth noting that in both cases (explicit proofs and type annotations) the verification method is type checking. For proof-carrying code, this is always

[1] This should not be confused with a digital signature used to certify authenticity.

type checking in the LF logical framework with some optimizations to eliminate redundant work. For typed assembly language the algorithm for type checking varies with the safety property that is enforced, although the basic nature of syntax-directed code traversal remains the same.

3 Automated Certification

How are certificates of safety to be obtained? In principle we may use any means at our disposal, without restriction or limitation. This freedom is assured by the checkability of safety certificates — it is always possible to determine mechanically whether or not a given certificate underwrites the safety of a given program. Since the code recipient can always check the validity of a safety certificate, there is no need to rely on the means by which the certificate was produced.

Two factors determine how hard it is to construct a safety certificate for a program:

(1) The strength of the assurances we wish to make about a program. The stronger the assurances, the harder it is to obtain a certificate.
(2) The complexity of the programming language itself. The more low-level the language, the harder it is to certify the safety of programs.

As a practical matter, the easier it is to construct safety certificates, the more likely that code certification will be widely used.

The main technique we have considered for building safety certificates is to build a *certifying compiler* for a safe, high-level language such as ML or Java (or any other type-safe language, such as Ada or Modula). A certifying compiler generates object code that is comparable (and often superior) in quality to that of an ordinary compiler. A certifying compiler goes beyond conventional compilation methods by augmenting the object code with a checkable safety certificate warranting the compliance of the object code with the safety properties of the source language. In this way we are able to exploit the safety properties of semantically well-defined high-level languages without having to trust the compiler itself or having to ensure the integrity of code in transit from producer to consumer.

The key to building a certifying compiler is to propagate *safety invariants* from the source language through the intermediate stages of compilation to the final object code. This means that each compilation phase is responsible for the preservation of these invariants from its input to its output. Moreover, to ensure checkability of these invariants, each phase must annotate the program with enough information for a code recipient to reconstruct the proof of these invariants. In this way the code recipient can check the safety of the code, without having to rely on the correctness of the compiler. In the (common) case that the compiler contains errors, the purported safety certificate may or may not be valid, but the recipient can detect the mistake. Since each compilation phase can be construed as a recipient of the code produced by the preceding stage, the compiler can check its own integrity by verifying the claimed invariants after each stage. This has proved to be an invaluable aid to the compiler writer [11,18].

3.1 Constructing Evidence of Safety

We have explored two main methods for propagating safety invariants during compilation:

(1) **Translation between typed intermediate languages** [26]. Safety invariants are captured by a type system for the intermediate languages of the compiler. The type system is designed to ensure that well-typed expressions are safe, and enough type information attached to intermediate forms to ensure that we may mechanically check type correctness. The typed intermediate forms are "self-certifying" in the sense that the attached type information serves as a checkable certificate of safety.
(2) **Compilation to proof-carrying code** [11]. Safety invariants are directly expressed as logical assertions about the execution behavior of conventional intermediate code. The soundness of the logic ensures that these assertions correctly express the required safety properties of the code. The safety of the object code is checked by a combination of verification condition generation and automatic theorem proving. By equipping the theorem prover with the means to generate a formal representation of a proof, we may generate checkable safety certificates for the object code.

These two methods are not mutually exclusive. We are currently exploring their integration using *dependent types* which allow assertions to be blended with types in a single type-theoretic formalism. This technique is robust and can be applied to high-level languages [27,28] as well as low-level languages [29], thereby providing an ideal basis for their use in certifying compilers.

To give a sense of how type information might be attached to intermediate code, we give an example derived from the representation of lists. At the level of the source language, there are two methods for creating lists:

(1) `nil`, which stands for the empty list;
(2) `cons(`h`, `t`)`, which constructs the non-empty list with head h and tail t.

These values are assigned types according to the following rules:[2]

(1) `nil` has type `list`;
(2) If h has type `int` and t has type `list`, then `cons(`h`,`t`)` has type `list`.

There are a variety of operations for manipulating lists, including the `car` and `cdr` operations, which have the following types:

(1) If l has type `list`, then `car(`l`)` has type `int`;
(2) If l has type `list`, then `cdr(`l`)` has type `list`.

The behavior of these operations is governed by the following transitions in an operational semantics for the language:

(1) `car(cons(`h`,`t`))` reduces to h;
(2) `cdr(cons(`h`,`t`))` reduces to t.

One task of the compiler is to decide on a representation of lists in memory, and to generate code for `car` and `cdr` consistently with this representation. A typical (if somewhat simple-minded) approach is to represent a list by

(1) A *pointer* to ...
(2) ... a *tagged* region of memory containing ...
(3) ... a *pair* consisting of the head and tail of the list.

The tag field distinguishes empty from non-empty lists, and the pointer identifies the address of the node in the heap. This representation can be depicted as the following compound term:

`ptr(tag[cons](pair(`h`, `t`)))`

What is interesting is that each individual construct in this expression may be thought of as a primitive of a typed intermediate language. Specifically,

[2] For simplicity we consider only lists of integers.

(1) `ptr(`v`)` has type `list` if v has type `[nil:void,cons:int*list]`. The bracketed expression defines the tags (`nil` and `cons`), and the type of their associated data values (none, in the case of `nil`, a pair in the case of a `cons`).
(2) `tag[t](`v`)` has type `[t:`τ`,...]` if v has type τ. In particular, `tag[cons](pair(`h`,`t`))` has type `[nil:void,cons:int*list]` if h has type `int` and t has type `list`.
(3) `pair(`l`,`r`)` has type τ_l`*`τ_r if l has type τ_l and r has type τ_r. In particular, `pair(`h`,`t`)` has type `int*list` if h has type `int` and t has type `list`.

Corresponding to this representation we may generate code for, say, `car(`l`)` that behaves as follows:

(1) Dereference the pointer l. The value l must be a pointer because its type is `list`.
(2) Check the tag of the object in the heap to ensure that it is `cons`. It must be either `cons` or `nil` because the type of the dereferenced pointer is `[nil:void,cons:int*list]`.
(3) Extract the underlying pair and project out its first component. It must have two components because the type of the tagged value is `int*list`.

When expressed formally in a typed intermediate language, the generated code for the `car` operation is defined in terms of primitive operations for performing these three steps. The safety of this code is ensured by the typing rules associated with these operations — a type correct program cannot misinterpret data by, for example, treating the head of a list as a floating point number (when it is, in fact, an integer).

A type-directed compiler [26] is one that performs transformations on typed intermediate languages, making use of type information to guide the translation, and ensuring that typing is preserved by each transformation stage. In a type-directed compiler each compilation phase translates not only the program code, but also its type, in such a way that the translated program has the translated type. How far this can be pushed is the subject of ongoing research. In the TILT compiler we are able to propagate type information down to the RTL (register transfer language level), at which point type propagation is abandoned. The recent development of Typed Assembly Language (TAL) [7,2] demonstrates the feasibility of propagating type information down to x86-like assembly code. The integration of TILT and TAL is the subject of ongoing research.

What does the propagation of type information have to do with safety? A well-behaved type system is one for which we can prove a *soundness theorem* relating the execution behavior of a program to its type. One consequence of the soundness theorem for the type system is that it is impossible for well-typed programs to incur type errors, memory errors, or control errors. That is, well-typed programs are safe. Of course not every safe program is well-typed — typing is a sufficient condition for safety, but not a necessary one. However, we may readily check type correctness of a program using lightweight and well-understood methods. The technique of type-directed compilation demonstrates that a rich variety of programs can be certified using typed intermediate languages. Whether there are demands that cannot be met using this method remains to be seen.

3.3 Logical Assertions and Explicit Proofs

Another approach to code certification that we are exploring [30,11] is the use of a combination of logical assertions and explicit proofs. A certifying compiler such as Touchstone works by augmenting intermediate code with logical assertions tracking the types and ranges of values. Checking the validity of these assertions is a two-step process:

(1) Verification condition generation (vcgen). The program is "symbolically evaluated" to propagate the implications of the logical assertions through each of the instructions in the program. This results in a set of logical implications that must hold for the program to be considered properly annotated.
(2) Theorem proving. Each of the implications generated during vcgen are verified using a combination of automatic theorem proving techniques, including constraint satisfaction procedures (such as simplex) and proof search techniques for first-order logic.

In this form the trusted computing base must include both the vcgen procedure and the theorem prover(s) used to check the verification conditions. In addition, a specification of a *safety policy* which describes the conditions for safe execution as well as pre- and post-conditions on all procedures supplied by the host operating system and required of the certified code.

To reduce the size of the trusted computing base, we may regard the combination of vcgen and theorem proving as a kind of "post-processing" phase in which the validity of the annotated program is not only checked, but a formal representation of the proof for the validity of the verification conditions is attached to the code. This is achieved by using *certifying theorem provers* [11,12] that not only seek to prove theorems, but also provide an explicit representation of the proof whenever one is found. Once the proofs have been obtained, it is much simpler to check them than it is to find them. Indeed, only the proof checker need be integrated into the trusted computing base; the theorem provers need not be trusted nor be protected from tampering.

To gain an understanding of what is involved here, consider the array subscript operation in a safe language. Given an array A of length n and an integer i, the operation sub(A,i) checks whether or not $0 \leq i < n$ and, if so, retrieves the ith element of A. At a high level this is an atomic operation, but when compiled into intermediate code it is defined in terms of more primitive operations along the following lines:

```
if (0 <= i && i < *A) {
   return A[i+1] /* unsafe access */
} else {
   ... signal an error ...
}
```

Note that *A refers to the length of array A. Here we assume that an integer array is represented by a pointer to a sequence of words, the first of which contains the array's length, and the rest of which are its contents.

Annotating this code with logical assertions, we obtain the following:

```
/* int i, array A */
if (0 <= i && i < *A) {
   /* 0 <= i < length(A) */
   return A[i+1]
} else {
   ... signal an error ...
}
```

The assertion that A is an array corresponds to the invariants mentioned above; in practice, a much lower-level type system is employed [11]. It is a simple matter to check that the given assertions are correct in this case.

Observe that the role of the conditional test is to enable the theorem prover to verify that the index operation A[i+1] is memory-safe — it does not stray beyond the bounds of the array. In many cases the run-time test is redundant because the compiler is able to *prove* that the run-time test must come out true, and therefore can be eliminated. For example, if the high-level code were a simple loop such as the following, we can expect the individual bounds checks to be elided:

```
int sum = 0;
for (i=0; i<length(A); i++) {
  /* 0 <= i < length(A) */
  sum += sub(A,i);
}
```

At the call site for `sub` the compiler is able to prove that $0 \leq i < n$, where n is the length of A. Propagating this through the code for `sub`, we find that the conditional test can be eliminated because the compiler can prove that the test must always be true. This leads to the following code (after further simplification):

```
int sum = 0;
for (i=0; i<*A; i++) {
  /* 0 <= i < length(A) */
  sum += A[i+1];
}
```

Given this annotation, we can now perform verification condition generation and theorem proving to check that the required precondition on the unsafe array subscript operation is indeed true, which ensures that the program is safe to execute. However, rather than place this additional burden on the programmer, we can instead attach a formal representation π of the proof of this fact to the assertions:

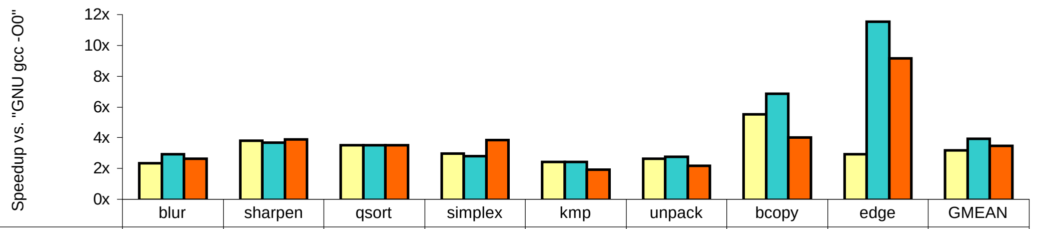
```
int sum = 0;
for (i=0; i<*A; i++) {
  /* π : 0 <= i < length(A) */
  sum += A[i+1]
}
```

The proof term π is a checkable witness to the validity of the given assertions that can be checked by the code recipient. In practice this witness is a term of the LF λ-calculus for which proof checking is simply another form of type checking (see Section 2.3).

4 Experimental Results

As mentioned earlier, we have implemented several systems to test and demonstrate the ideas of certified code, typed intermediate languages, certifying compilers, and certifying theorem provers. The results of our experiments with these systems confirm several important claims about the general framework for safety certification of code that we have presented in this paper.

(1) Approaches to certified code such as PCC and TAL allow highly optimized code to be verified for safety. This means that few if any compromises need to be made between high performance and safety.
(2) The various approaches to certifying compilers that we have explored, such as typed intermediate languages and logical assertions, can be scaled



	blur	sharpen	qsort	simplex	kmp	unpack	bcopy	edge	GMEAN
🟨 GNU gcc -O4	2.33	3.82	3.51	2.97	2.44	2.62	5.50	2.92	3.17
🟦 DEC cc -O4	2.92	3.68	3.52	2.79	2.44	2.76	6.88	11.52	3.92
🟧 Cert Comp	2.64	3.89	3.52	3.86	1.93	2.20	4.00	9.16	3.48

Fig. 1. Comparison of generated object-code performance between the Touchstone, GCC, and DEC CC optimizing compilers. The height of the bars shows the speedup of the object code relative to unoptimized code as produced by gcc.

up to languages of realistic scale and complexity. Furthermore, they provide an automatic means of obtaining code that is certified to hold standard safety properties such as type safety, memory safety, and control safety.

(3) The need to include annotations and/or proofs with the code is not an undue burden. Furthermore, checking these certificates can be performed quickly and reliably.

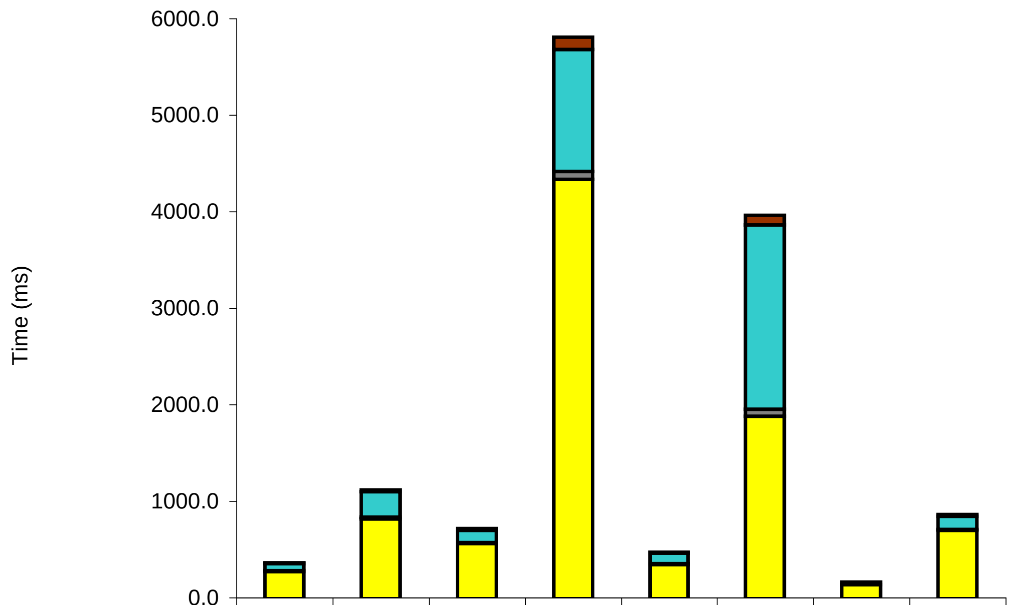
In order to support these claims and give a better feel for the practical details in our systems, we now present some results of our experiments.

4.1 The Touchstone Certifying Compiler

Touchstone is a certifying compiler for an imperative programming language with a C-like syntax. Although the source programs look very much like C programs, the language compiled by Touchstone is made "safe" by having a strong static type system, eliminating pointer arithmetic, and ensuring that all variables are initialized. Although this language makes restrictions on C, it is still a rich and powerful language in the sense of allowing recursive procedures, aliased variables, switch statements, and dynamically allocated data structures. Indeed, it is straightforward to translate many practical C source programs into the language compiled by Touchstone [31].

Given such a source program, Touchstone generates a highly optimized native code target program for the DEC Alpha architecture with an attached proof of its type, memory, and control safety.

Figure 1 shows the results of a collection of benchmark programs when compiled with Touchstone, the Gnu gcc C compiler, and the DEC cc C compiler. The benchmark programs were obtained from standard Unix utility applica-



	blur	sharpen	qsort	simplex	kmp	unpack	bcopy	edge
🟫 Proof Checking (ms)	7.7	23.5	16.3	120.3	9.6	92.5	3.6	14.7
🟦 Proving (ms)	81.0	257.0	127.0	1272.0	108.0	1912.0	25.0	143.0
⬛ VC Generation (ms)	6.3	20.9	11.5	73.9	8.4	70.6	3.5	9.7
🟨 Code Generation (ms)	271.0	818.0	560.0	4340.0	348.0	1885.0	136.0	697.0

Fig. 2. Breakdown of time required to generated the proof-carrying code binaries.

tions (such as the xv and gzip programs) and then edited in a completely straightforward way to replace uses of pointer arithmetic with array-indexing syntax. (Recall that the C-like language compiled by Touchstone does not support pointer arithmetic.) The bars in the figure were generated by first compiling each program with the Gnu gcc compiler with all optimizations turned off. Then, Touchstone, Gnu gcc, and DEC cc were used to compile the programs with all optimizations turned on. The bars in the figure show the relative speed improvements produced by each optimizing compiler relative to the unoptimized code.

The figure shows that the Touchstone compiler generates object code which is comparable in speed to that produced by the gcc and cc compilers, and in fact is superior to gcc overall. This result is particularly surprising when one considers that Touchstone is obligated to guarantee that all array accesses and pointer dereferences are safe (that is, Touchstone must sometimes perform array-bounds and null-pointer checks), whereas the gcc and cc compilers do not do this. In fact, Touchstone is able to optimize away almost all array-bounds and null-pointer checks, and generates proofs that can convince any code recipient that all array and pointer accesses are still safe.

In Figure 2 we provide a breakdown of the time required to compile each benchmark program into a PCC binary. Each bar in the figure is divided into four parts. The bottom-most part shows the “conventional” compile time. This is the time required to generate the DEC Alpha assembly code plus invariant annotations required by the underlying PCC system. Because Touchstone is a highly aggressive optimizing compiler, it is a bit slower than typical compilers. However, on average it is comparable in compiling times to the DEC cc compiler with all optimizations enabled. The second part shows the time required to generate the verification conditions. Finally, the third and fourth parts show the times required for proof generation and proof checking, respectively.

One can see that very little time is required for the verification-condition generation and proof checking. This is important because it is these two steps that must also be performed by any recipient of the generated code. The fact that these two parts are so small is an indication that the code recipient in fact has very little work to do.

Early measurements with the Touchstone compiler showed that the proofs were about 2 to 4 times larger than the code size [31]. Since the time that those experimental results were obtained, we have made considerable progress on reducing the size of the proofs, without increasing the time or effort required to check them. These reductions lead to proof sizes on the order of 10% to 40% of the size of the code. In addition, we have been experimenting with a new representation (which we refer to as the “oracle string” representation) which, for the types of programs described here, further reduces proof sizes to be consistently less than 5% of the code size, at the cost of making proof checking about 50% slower. We hope to be able to describe these techniques and show their effects in a future report.

4.2 The Cedilla Systems Special J Compiler

The experimental results shown above are admittedly less than convincing, due to the relatively small size of the test programs. Recently, however, we have “spun off” a commercial enterprise to build an industrial-strength implementation of a proof-carrying code system. This enterprise, called Cedilla Systems Incorporated, is essentially an experiment in technology transfer, in the sense that it is attempting to take ideas and results directly out of the laboratory and into commercial practice. Cedilla Systems has shown that the ideas presented in this paper can be scaled up to full-scale languages. This is shown most clearly in an optimizing native-code compiler for the full Java programming language, called Special J [32], which successfully compiles over 300 “real-world” Java applications, including rather large ones such as Sun’s StarOffice application suite and their HotJava web browser.

The operation of Special J is similar to Touchstone, in that Special J produces optimized target code annotated with invariants that make it possible to construct a proof of safety. A verification-condition generator is then used to extract a verification condition, and a certifying theorem prover generates the proof which is attached to the target code.

To see a simple example of this process, consider the following Java program:

```
public class Bcopy1 {
 public static void bcopy(int[] src,
                          int[] dst)
 {
  int l = src.length;
  int i = 0;
  for(i=0; i<l; i++) {
    dst[i] = src[i];
  }
 }
}
```

This source program is compiled by Special J into the target program for the Intel x86 architecture shown in Figure 3. Included in this target program are numerous data structures to support Java's object model and run-time system. The core of this output, however, is the native code for the `bcopy` method shown above.

This code is largely conventional except for the insertion of several invariants, each of which is marked with a special "`ANN_`" macro. These annotations are "hints" from the compiler that help the automatic proof generator do its job. They do not generate code, and they do not constrain the object code in any way. However, they serve an important engineering purpose, as we will now describe.

The `ANN_LOCALS` annotation simply says that the compiled method uses three locals. In this case, the register allocator did not need any spill space on the stack, so the only locals are the two formal parameters and the return address. This hint is useful for proving memory safety. The prover could, in principle, analyze the code itself to reverse-engineer this information; but it is much easier for the compiler to communicate what it already knows. Since one of our engineering goals is to simplify as much as possible the size of the trusted computing base, it is better to have the compiler generate this information, leaving only the checking problem to the PCC infrastructure.

The `ANN_UNREACHABLE` annotations come from the fact that the safety policy specifies that array accesses must always be in bounds and null pointers must never be dereferenced. In Java, such failures result in run-time exceptions, but

```
ANN_LOCALS(_bcopy__6arrays6Bcopy1AIAI, 3)
.text
.align 4
.globl _bcopy__6arrays6Bcopy1AIAI
_bcopy__6arrays6Bcopy1AIAI:
  cmpl  $0, 4(%esp)   ;src==null?
  je    L6
  movl  4(%esp), %ebx
  movl  4(%ebx), %ecx ;l = src.length
  testl %ecx, %ecx    ;l==0?
  jg    L22
  ret
L22:
  xorl  %edx, %edx    ;initialize i
  cmpl  $0, 8(%esp)   ;dst==null?
  je    L6
  movl  8(%esp), %eax
  movl  4(%eax), %esi ;dst.length
L7:
ANN_INV(ANN_DOM_LOOP,
  LF_(/\ (csubneq ebx 0)
     (/\ (csubneq eax 0)
     (/\ (csubb edx ecx)
        (of rm mem))))_LF,
  RB(EDI,EDX,EFLAGS,FFLAGS,RM))
  cmpl  %esi, %edx    ;i<dst.length?
  jae   L13
  movl  8(%ebx, %edx, 4), %edi ;src[i]
  movl  %edi, 8(%eax, %edx, 4) ;dst[i]=
  incl  %edx          ;i++
  cmpl  %ecx, %edx    ;i<l?
  jl    L7
  ret
ANN_INV(ANN_DOM_LOOP,
        LF_true_LF,
        RB(FFLAGS))
  ret
L13:
  call  __Jv_ThrowBadArrayIndex
ANN_UNREACHABLE
  nop
L6:
  call  __Jv_ThrowNullPointer
ANN_UNREACHABLE
  nop
```

Fig. 3. Special J output code

the safety policy in our example requires a proof that these exceptions will never be thrown. Therefore, the compiler points out places that must never be reached during execution so that the proof generator does not need to reverse-engineer where the source-code array accesses and pointer dereferences ended up in the binary.

The first `ANN_INV` annotation is by far the most interesting of all the annotations. Note that the Special J compiler has optimized the tight loop:

- Both required null checks are hoisted. (Note that the null check on `dst` cannot be hoisted before the loop entry because the loop may never be entered at all; but it can be hoisted to the first iteration.)
- The bounds check on `src` is hoisted. (Note that hoisting the bounds check on `dst` would be a more exotic optimization, because in the case that `dst` is not long enough, the loop must copy as far as it can and then throw an exception.)

The proof generator must still prove memory safety, so it must prove that inside the loop there are no null-pointer dereferences or out-of-bounds memory accesses. Essentially, the proof generator needs to go through the same reasoning that the compiler went through when it hoisted those checks outside the loop. Therefore, to help the proof generator, the compiler outputs the relevant loop invariants that it discovered while performing the code-hoisting optimizations. In this case, it discovered that:

- `src` (in register `ebx`) is not null: `(csubneq ebx 0)`.
- `dst` (in register `eax`) is not null: `(csubneq eax 0)`.
- `i` (`edx`) is unsigned-below `src.length` (`ecx`): `(csubb edx ecx)`.

The "`csub`" prefix denotes the result of a Pentium comparison. Other things in the loop invariant specify:

- which registers are modified in the loop: `RB(...)`,
- that memory safety is a loop invariant: `(of rm mem)`

Pseudo-register `rm` denotes the computer's memory, and `(of rm mem)` means that no unsafe operations have been performed on the memory.

After this target code is generated by Special J, the Cedilla Systems proof generator reads it and outputs a proof that the code satisfies the safety policy. The first step to doing this is to generate a logical predicate, called a *verification condition* (or simply VC), whose logical validity implies the safety of the code. It is important that the same VC be used by both the producer and the recipient of the code, so that the recipient can guarantee that the "right" safety proof is provided, as opposed to a proof of some unrelated or irrelevant property.

As we explained earlier, both the proofs and the verification conditions are expressed in a language called the Logical Framework (LF). Space prevents us from including the entire VC for our bcopy example. However, the following excerpt illustrates the main points. (Note that `X0` is the `dst` parameter, `X1` is the `src` parameter, `X2` is a pseudo-register representing the current state of the heap, and `X3` is the variable `i`.

```
(=> (csubb X3 (sel4 X2 (add X1 4)))
 (=> (csubneq X0 0)
  (=> (csubneq X1 0)
   (=> (csubb X3 (sel4 X2 (add X0 4)))
    (/\ (saferd4
          (add X1 (add (imul X3 4) 8)))
     (/\ (safewr4
          (add X0 (add (imul X3 4) 8))
         (sel4 X4
          (add X1 (add (imul X3 4) 8))))
      (/\
       (=> (csublt (add X3 1)
                   (sel4 X2 (add X1 4)))
        (/\ (csubneq X1 0)
         (/\ (csubneq X0 0)
          (/\ (csubb (add X3 1)
                     (sel4 X2 (add X1 4)))
```

This excerpt of the VC says that, given the loop-invariant assumptions

- `(csubb X3 (sel4 X2 (add X1 4)))` (i.e., `src[i]` is in bounds),
- `(csubneq X0 0)` (i.e., `dst` is non-null), and
- `(csubneq X1 0)` (i.e., `src` is non-null),

and given the bounds check that was emitted for `dst`:

- `(csubb X3 (sel4 X2 (add X0 4)))`

as well as some additional assumptions outside the loop (not shown in this snippet), proofs are required to establish the safety of the read of the `src` array and the write to the `dst` array. Furthermore, given the additional loop-entry condition

- `(csublt (add X3 1) (sel4 X2 (add X1 4)))`

proofs are required to reestablish the loop invariants.

Here, `X0` corresponds to `eax` (`dst` in the source), `X1` to `ebx` (`src` in the source), X2 to rm (the memory pseudo-register), and X3 to edx (i in the source). Note that src.length is (sel4 X2 (add X1 4)), because the length is stored at byte-offset 4 in an array object. The safety policy, and hence the VC, specifies and enforces these requirements on data-structure layout.

The proof generator reads the VC and outputs a proof of it. A tiny excerpt of this proof is shown below:

```
(impi
 ([ASS10: pf (csubb X3
             (sel4 X2 (add X1 4)))]
  (impi
   ([ASS11: pf (csubneq X0 0)]
    (impi
     ([ASS12: pf (csubneq X1 0)]
      (impi
       ([ASS13: pf (csubb X3
                    (sel4 X2
                     (add X0 4)))]
        (andi
         (rdArray4 ASS4 ASS3
                   (sub0chk ASS12)
                   szint
                   (aidxi 4
                    (below1 ASS10)))
 ...
```

The proofs are shown here in a concrete syntax for LF developed for the Elf system [22,21]. In this very small snippet of the proof, one can see that assumptions (marked with the "ASS..." identifiers) are labeled and then used in the body of the proof. Logical inference rules such as "impi", which in this case stands for the "implication-introduction" rule, are specified declaratively in the LF language, and included with the PCC system as part of the definition of the safety policy.

Finally, a binary encoding of the proof is made and attached to the target code. The proof is included in the data segment of a standard binary in the COFF format. In this case, the proof takes up 7.1% of the total object file. We note that we currently use an unoptimized binary encoding of the proof in which all proof tokens are 16 bits long. Huffman encoding produces an average token size of 3.5 bits, and so a Huffman-encoded binary is expected to be about 22% of the size of a non-Huffman-encoded binary. In this case, that would make the size of the proof approximately 45 bytes, or less than 2% of the object file. While Huffman encoding would indeed be an effective means of reducing the size of proofs, we have found that other representations such as "oracle strings" do an even better job, without incurring the cost of decompression. In the case of the current example, the oracle string representation of the proof requires less than 6 bytes. We hope to report in detail on this representation in a future report.

5 Conclusion and Future Work

We have presented a general framework for the safety certification of code. It relies on the formal definition of a safety policy and explicit evidence for compliance attached to mobile code. This evidence may take the form of formal safety proofs (in *proof-carrying code*) or type annotations (in *typed assembly language*). In both cases one can establish with mathematical rigor that certified code is tamper-proof and can be executed safely without additional run-time checks or operating system protection boundaries. Experience with the approaches has shown the overhead to be acceptable in practice, both in the time to validate the certificate and the space to represent it, using advanced techniques from logical frameworks and type theory.

We also sketched how certificates can be obtained automatically through the use of certifying compilers and theorem provers. The approach of *typed intermediate languages* propagates safety properties which are guaranteed for the high-level source language throughout the compilation process down to low-level code. Safety remains verifiable at each layer through type-checking. A *certifying compiler* such as Touchstone uses logical assertions throughout compilation in a similar manner, except that the validity of the logical assertions must be assured by theorem proving. This is practical for the class of safety policies considered here, since the compiler can provide the information necessary to guarantee that a proof can always be found. Finally, a *certifying theorem prover* does not need to be part of the trusted computing base since it produces explicit proof terms which can be checked independently by an implementation of a logical framework.

The key technology underlying our approaches to safety is type theory as used in modern programming language design and implementation. The idea that type systems guarantee program safety and modularity for high-level languages is an old one. We see our main contribution in demonstrating in practical, working systems such as Touchstone, the TILT compiler, and the Twelf logical framework, that techniques from type theory can equally be applied to intermediate and low-level languages down to machine code in order to support provably safe mobile code for which certificates can be generated automatically.

References

[1] R. Wahbe, S. Lucco, T. Anderson, S. Graham, Efficient software-based fault isolation, in: 14th ACM Symposium on Operating System Principles, 1993, pp. 203–216.

[2] G. Morrisett, K. Crary, N. Glew, D. Walker, Stack-based typed assembly language, in: X. Leroy, A. Ohori (Eds.), Second Workshop on Types in Compilation, Vol. 1473 of Lecture Notes in Computer Science, Springer-Verlag, Kyoto, Japan, 1998, pp. 28–52, extended version published as CMU technical report CMU-CS-98-178.

[3] G. Necula, P. Lee, Safe, untrusted agents using proof-carrying code, in: Special Issue on Mobile Agent Security, Vol. 1419 of Lecture Notes in Computer Science, Springer-Verlag, 1997.

[4] K. Crary, S. Weirich, Resource bound certification, in: Twenty-Seventh ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, 2000, pp. 184–198.

[5] D. Walker, A type system for expressive security policies, in: Twenty-Seventh ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, 2000, pp. 254–267.

[6] G. Necula, P. Lee, Safe kernel extensions without run-time checking, in: Second Symposium on Operating Systems Design and Implementation, Seattle, Washington, 1996, pp. 229–243.

[7] G. Morrisett, D. Walker, K. Crary, N. Glew, From System F to typed assembly language, ACM Trans. Prog. Lang. Syst. 21 (3) (1999) 528–569, an earlier version appeared in the Twenty-Fifth Symposium on Principles of Programming Languages (POPL'98), San Diego, California, 1998, pp. 85–97.

[8] R. Harper, F. Honsell, G. Plotkin, A framework for defining logics, Journal of the Association for Computing Machinery 40 (1) (1993) 143–184.

[9] R. Harper, F. Pfenning, On equivalence and canonical forms in the LF type theory, Tech. Rep. CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University (Jul. 2000).

[10] R. Virga, Higher-order rewriting with dependent types, Ph.D. thesis, Department of Mathematical Sciences, Carnegie Mellon University, available as Technical Report CMU-CS-99-167 (Sep. 1999).

[11] G. C. Necula, Compiling with proofs, Ph.D. thesis, Carnegie Mellon University, available as Technical Report CMU-CS-98-154 (Oct. 1998).

[12] A. Stump, D. L. Dill, Generating proofs from a decision procedure, in: A. Pnueli, P. Traverso (Eds.), Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy, 1999.

[13] S. Michaylov, F. Pfenning, An empirical study of the runtime behavior of higher-order logic programs, in: D. Miller (Ed.), Proceedings of the Workshop on the λProlog Programming Language, University of Pennsylvania, Philadelphia, Pennsylvania, 1992, pp. 257–271, available as Technical Report MS-CIS-92-86.

[14] G. C. Necula, P. Lee, Efficient representation and validation of logical proofs, in: Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98), IEEE Computer Society Press, Indianapolis, Indiana, 1998, pp. 93–104.

[15] F. Pfenning, C. Schürmann, Algorithms for equality and unification in the presence of notational definitions, in: T. Altenkirch, W. Naraschewski, B. Reus (Eds.), Types for Proofs and Programs, Springer-Verlag LNCS 1657, Kloster Irsee, Germany, 1998, pp. 179–193.

[16] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, S. Eggers, Extensibility, safety and performance in the SPIN operating system, in: Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain, 1995, pp. 267–284.

[17] J. C. Reynolds, Definitional interpreters for higher-order programming languages, in: Conference Record of the 25th National ACM Conference, Boston, 1972, pp. 717–740.

[18] J. G. Morrisett, Compiling with types, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–95–226.) (December 1995).

[19] A. W. Appel, E. W. Felten, Proof-carrying authentication, in: G. Tsudik (Ed.), Proceedings of the 6th Conference on Computer and Communications Security, ACM Press, Singapore, 1999, pp. 52–62.

[20] A. W. Appel, A. P. Felty, A semantic model of types and machine instructions for proof-carrying code, in: T. Reps (Ed.), Conference Record of the 27th Annual Symposium on Principles of Programming Languages (POPL'00), ACM Press, Boston, Massachusetts, 2000, pp. 243–253.

[21] F. Pfenning, C. Schürmann, System description: Twelf — a meta-logical framework for deductive systems, in: H. Ganzinger (Ed.), Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Springer-Verlag LNAI 1632, Trento, Italy, 1999, pp. 202–206.

[22] F. Pfenning, Elf: A meta-language for deductive systems, in: A. Bundy (Ed.), Proceedings of the 12th International Conference on Automated Deduction, Springer-Verlag LNAI 814, Nancy, France, 1994, pp. 811–815, system abstract.

[23] F. Pfenning, Logic programming in the LF logical framework, in: G. Huet, G. Plotkin (Eds.), Logical Frameworks, Cambridge University Press, 1991, pp. 149–181.

[24] F. Pfenning, Logical frameworks, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science and MIT Press, 2001, Ch. XXI, in preparation.

[25] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic, TALx86: A realistic typed assembly language, in: Second Workshop on Compiler Support for System Software, Atlanta, Georgia, 1999, pp. 25–35.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee, TIL: A type-directed optimizing compiler for ML, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, 1996, pp. 181–192.

[27] H. Xi, F. Pfenning, Eliminating array bound checking through dependent types, in: K. D. Cooper (Ed.), Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98), ACM Press, Montreal, Canada, 1998, pp. 249–257.

[28] H. Xi, F. Pfenning, Dependent types in practical programming, in: A. Aiken (Ed.), Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99), ACM Press, 1999, pp. 214–227.

[29] H. Xi, R. Harper, A dependently typed assembly language, Tech. Rep. OGI-CSE-99-008, Computer Science Department, Oregon Graduate Institute (July 1999).

[30] G. C. Necula, Proof-carrying code, in: N. D. Jones (Ed.), Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97), ACM Press, Paris, France, 1997, pp. 106–119.

[31] G. C. Necula, P. Lee, The design and implementation of a certifying compiler, in: K. D. Cooper (Ed.), Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98), ACM Press, Montreal, Canada, 1998, pp. 333–344.

[32] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, K. Cline, A certifying compiler for Java, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00), ACM Press, Vancouver, Canada, 2000, pp. 95–107.