Language-Based Security

Dexter Kozen

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501, USA
kozen@cs.cornell.edu

Abstract. Security of mobile code is a major issue in today's global computing environment. When you download a program from an untrusted source, how can you be sure it will not do something undesirable? In this paper I will discuss a particular approach to this problem called *language-based security*. In this approach, security information is derived from a program written in a high-level language during the compilation process and is included in the compiled object. This extra security information can take the form of a formal proof, a type annotation, or some other form of certificate or annotation. It can be downloaded along with the object code and automatically verified before running the code locally, giving some assurance against certain types of failure or unauthorized activity. The verifier must be trusted, but the compiler, code, and certificate need not be. Java bytecode verification is an example of this approach. I will give an overview of some recent work in this area, including a particular effort in which we are trying to make the production of certificates and the verification as efficient and invisible as possible.

1 Introduction

With the rise of the Internet, security of mobile code is emerging as one of the most important challenges facing computing research today. As we become more and more dependent on the global information infrastructure, we are finding ourselves increasingly vulnerable to malicious attacks and buggy software. Yet, even as Melissa and Happy99 wreak worldwide havoc, we continue to download and run plug-in software with little regard for the consequences.

A recent study of the Computer Science and Telecommunications Board of the National Research Council [30] details the extent of the security problem. It argues that much of our critical infrastructure—transportation, communication, financial markets, energy distribution, health care—is becoming dangerously dependent on a computing base that is out of the purview of any single authority. We are already vulnerable to many forms of malicious attack and software failure with potentially devastating consequences.

The recent report of the President's Information Technology Advisory Committee (PITAC) [10] warns of this dependence and recommends a substantial increase in federally funded research:

> We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways ... Our nation's dependence on the Internet is increasing. While this is an exciting development, the Internet is growing well beyond the intent of its original designers and our ability to extend its use has created enormous challenges. As the size, capability, and complexity of the Internet grows, it is imperative that we do the necessary research to learn how to build and use large, complex, highly-reliable, and secure systems ... This research will ... protect us from catastrophic failures of the complex systems that now underpin our transportation, defense, business, finance, and healthcare infrastructures. [10]

President Clinton and Vice President Gore responded to an interim version of this report with a far-reaching initiative known as IT^2 in which they propose a $366 million, 28% increase for research in information technology as part of the fiscal 2000 federal budget [20]. In their words:

> As our economy and society become increasingly dependent on information technology, we must be able to design information systems that are more secure, reliable, and dependable. The software systems that lie at the core of worldwide financial systems, air traffic management, defense command and control—indeed, virtually all parts of our economy—are the most complex human inventions ever created. As a result, however, our society now faces unknown hazards both from hostile attacks on these systems and from the even greater threat that simple mistakes or system failures will bring wholesale collapse of critical systems. The small software failures that have shut down large parts of the nation's phone systems and air traffic control systems and the "millennium bug" are examples of what can go wrong in our current environment. We do not know how to design and test complex software systems with millions of lines of code in the same way that we can verify whether a bridge or an airplane is safe. [20, p. 5]

And from elsewhere in the same report:

> Active software participates in its own development and deployment. We see the first steps towards active software with "applets" that can be downloaded from the Internet, but this is just the beginning. Active software will eventually be able to update itself, monitor its progress toward a particular goal, discover a new capability that is needed for the task at hand, and safely and securely download the piece of software needed to perform that task. [20, p. 7]

In this paper we focus on a particular approach to the security problem known as *language-based security*. We give a general overview, then we discuss several current research projects that fall within this framework: proof-carrying code (PCC) [21–26], typed assembly language (TAL) [7, 13, 14, 16], security automata (SASI) [6, 28, 29], efficient code certification (ECC) [11], and information flow (JFlow) [17–19].

2 Some Issues in Security

2.1 Safety Policies

Suppose we wish to download and run a program from an unknown or untrusted source. Before running our downloaded program, it would be nice to have some assurance that the code is safe to run. Of course, "safe" is subject to interpretation and may have different meanings in different contexts. The definition of "safe" used in a particular application is that application's *safety policy*. For example, we may wish to be sure that the program will never accidentally overwrite critical system data, thereby causing a crash; this would be desirable in, say, flight control software or active messages in network routers. We may wish to know that the program does not access memory allocated to other processes running in the same physical address space; this would be an important consideration in smart cards. We may wish to deny all disk I/O; this is currently part of the default safety policy for Java applets downloaded off the net. This restriction is rather strong, and we may wish to weaken it to allow restricted forms of disk I/O. For example, we may wish to allow the applet to read disk files provided it does not send any messages out on the net afterwards, or we may wish to let it deposit a limited amount of data of a certain form (cookies) in a particular directory.

At the very minimum, any safety policy for untrusted machine code executing locally should guarantee the following fundamental safety properties.

- *Control flow safety.* The program should never execute a jump or call to a random location, but only addresses within its own code segment containing valid instructions. All calls should be to valid function entry points and all returns to the location from which the function was called.
- *Memory safety.* The program should not access random places in memory, but only valid locations in its own static data segment, live system heap memory explicitly allocated to it, and valid stack frames.
- *Stack safety.* For stack-based runtime architectures, the runtime stack should be preserved across function calls. We interpret this flexibly; minor modifications near the top of the stack are allowed, as is tail recursion elimination.

These three considerations turn out to be interdependent. The level of security they mutually represent is evidently the minimum nontrivial level of safety one could expect in the sense that it is hard to imagine a meaningful security policy that would be enforceable without them. More complicated policies are certainly possible depending on the application at hand.

Many papers consider *type safety* as well, however this makes sense only in the presence of a typing discipline. A typing discipline is a way of assigning *intention* to raw data and code. The type of a function usually gives a relationship between the input and output states in terms of this intention. For example, we might have a function of type $r_1 : \mathrm{int} \times r_2 : \mathrm{int} \to r_3 : \mathrm{int}$, which says that if registers r_1 and r_2 contain integer values when the code is called, then upon return, register r_3 must contain a valid integer. In the presence of types, control flow safety, memory safety, and stack safety are subsumed by type safety, since these properties are encoded in the typing discipline. This is the approach of TAL (see Section 4.3 below).

2.2 Trust

Security is based on the notion of *trust*. In reality, there may be different degrees of trust, but for the sake of simplicity we partition the universe into two classes, those agents and artifacts that are trusted and those that are not, separated by an imaginary *trust boundary*. All trusted software—that is, all software on our side of the trust boundary—is called our *trusted code base*.

All software security mechanisms depend on some trusted code. We can assume that the trusted code base includes at least the local operating system kernel and some programming language runtime support (provided they have not been corrupted; part of the security problem is to prevent this from happening). However, it is generally desirable to keep the trusted code base as small as possible, simply because the less we need to trust, the less vulnerable we are.

Control flow safety, memory safety, and stack safety can be guaranteed by writing the program in a type-safe language and compiling with a trusted compiler. The disadvantage of this solution is that the compiler must be part of the trusted code base. Either you must send me your source code so that I can compile it locally, which forces you to release proprietary source code and forces me to spend time compiling it, or I must trust your compiler and the channel by which you ship the object code to me. Both are unsatisfactory, the former from a performance standpoint and the latter from a security standpoint. Many of the approaches described below, including the language-based approach, achieve their objectives without assuming that the compiler is part of the trusted code base.

2.3 Performance vs. Safety

It is of course nice to have both strong safety guarantees and good performance, but these are often in conflict: the latter prefers to allow, the former to restrict. In a sense, much of the current research in security is concerned with resolving the tension between these opposing forces in some acceptable way. Different approaches to the security problem fall on different points of the spectrum, and it is perhaps unreasonable to expect a single mechanism to be optimal on both counts. This is true also for language-based mechanisms. However, language-based approaches can give improvements on both counts, as described in Section 4 below.

3 Traditional Approaches

Traditional approaches to the security problem include kernel as reference monitor, cryptography, code instrumentation, and trusted compilation. These mechanisms offer a fixed set of basic safety policies with little flexibility. Security automata and PCC are more recent developments that provide a general framework for expressing and enforcing a wide range of security policies. TAL and ECC share the same goals, but sacrifice expressiveness for efficiency.

Kernel as reference monitor is probably the oldest and most widespread security mechanism used in software systems. It refers to the practice of isolating operations on critical system components and data in a *system kernel*. The kernel is a privileged body of code that may access these critical components and data directly. All other processes may only access them in limited ways using the kernel as proxy, communicating their desires by message. This not only prevents untrusted code from corrupting the system, but also allows the kernel to monitor all access, perform authentication, or enforce other safety policies.

However, allowing non-kernel processes direct access to critical system components and data can improve performance significantly. With kernel calls, access is limited to a few high-level abstract operations provided by the kernel interface; but with direct access, more sophisticated algorithms that exploit properties of the low-level data structures can be used. Also, kernel calls typically involve some overhead for packaging parameters and for saving and restoring registers (called a *context switch*), which can be circumvented with direct access. It is therefore desirable to figure out how to allow non-kernel processes more direct access to critical system components and data without compromising security. The SPIN system [2] is one effort in this direction. The SPIN system enforces security by using a trusted compiler.

Cryptography can discourage access to sensitive data during transit across an untrusted network and can be used for authentication. Unfortunately, the safety of current cryptographic protocols depends on unproven complexity-theoretic assumptions. Current standards such as the Digital Encryption Standard (DES) can be broken by an agent with sufficient computing power [5]. To make matters worse, we are not completely free to use it; current policy regarding the commercial use of strong cryptography is hopelessly entangled in a web of political and legal complications [5]. Finally, cryptography alone cannot ensure that downloaded code is safe to run, only that it came from a particular source and that it has not been compromised in transit.

Code instrumentation refers to the process of altering (instrumenting) machine code so that critical operations can be monitored during execution. This is done in such a way that (i) the functional behavior of the instrumented code is the same as the original uninstrumented code, provided the original code would not have violated the safety policy; and (ii) if the original uninstrumented code would have violated the safety policy, then at the time of the violation, the instrumented code either detects the violation and causes the system to intercept control and shut down the errant process, or otherwise prevents the transgression from having any ill effects on the rest of the system.

An example of code instrumentation is *software fault isolation* (SFI) or *sandboxing* [32]. In one particularly simple and efficient variant of SFI, the untrusted code and data are loaded into a block of contiguous memory with addresses in the range $[c2^k, c2^k + 2^k - 1]$ for some integers c and k (the "sandbox") and then linked. Then a pass is made over the code, replacing the higher order $wordlength - k$ bits of all direct memory access and jump addresses with the bits of c. For indirect addresses, code is inserted to do this operation at runtime. This has no effect on instructions that target addresses inside the sandbox, so a correct program will not be affected. However, addresses outside the sandbox get mapped to addresses inside the sandbox. The addresses they get mapped to are random from the point of view of the program, which of course breaks the program, but the error is confined to the sandbox and cannot compromise the rest of the system.

Schneider [28, 29] extends this idea to handle any safety policy that can be expressed by a finite-state automaton. For example, one can express the condition, "No message is ever sent out on the net after a disk read," with a two-state automaton. These automata are called *security automata*. The code is instrumented so that every instruction that could potentially affect the state of the security automaton is preceded by a call to the automaton. Security automata give considerable flexibility in the specification of safety policies and allow the construction of specialized policies tailored to a consumer's particular needs. The main drawback is that some runtime overhead is incurred for the runtime calls to simulate the automaton.

An advantage of code instrumentation is that it can be performed in isolation by the consumer with no particular assumptions or extra information about the code. However, enforcing safety policies for arbitrary code by instrumentation alone can be costly. A runtime check is required before every sensitive operation, which could contribute substantially to runtime overhead. Some runtime checks can be eliminated if program analysis determines that they are unnecessary, but this is also costly undertaking and could contribute substantially to loadtime overhead. Moreover, even the most sophisticated analysis techniques are necessarily incomplete, because safety properties are undecidable in general.

There is recent evidence that code instrumentation can be used in conjunction with language-based methods to improve performance [6, 33].

4 Language-Based Security

Schneider defines *language-based security* very broadly as "a set of techniques based on programming language theory and implementation, including semantics, types, optimization, and verification, brought to bear on the security question." By that definition, SFI and SASI are instances of language-based security. For the purposes of this paper, however, we would like to focus on a more specific model.

Compilers for high-level programming languages typically accumulate much information about a program during the compilation process. This information may take the form of type information or other constraints on the values of variables, structural information, or naming information. This information may be obtained through parsing or program analysis and may be used to perform optimizations or to check type correctness. After a successful compilation, compilers traditionally throw this extra information away, leaving a monolithic sequence of instructions with no apparent structure or discernable properties.

However, some of this extra information may have implications regarding the safety of the compiled object code. For example, programs written in type-safe languages must typecheck successfully before they will compile, and assuming that the compiler is correct, any object code compiled from a successfully type-checked source program should be memory-safe. If a code consumer only had access to the extra information known to the compiler when the program was compiled, it might be easier to determine whether the downloaded object code is safe to run.

We will use the phrase *language-based security* to refer to the idea of retaining this extra information from a program written in a high-level language in the object code compiled from it. This extra information—call it a *certificate*—is created at compile time and packaged with the object code. When the code is downloaded, the certificate is downloaded along with it. The consumer can then run a *verifier*, which inspects the code and the certificate to verify compliance with a safety policy. If it passes the test, then the code is safe to run. The verifier is part of the consumer's trusted code base; the compiler, the compiled code, and the certificate need not be. Figure 1 illustrates a simplified version of this framework.

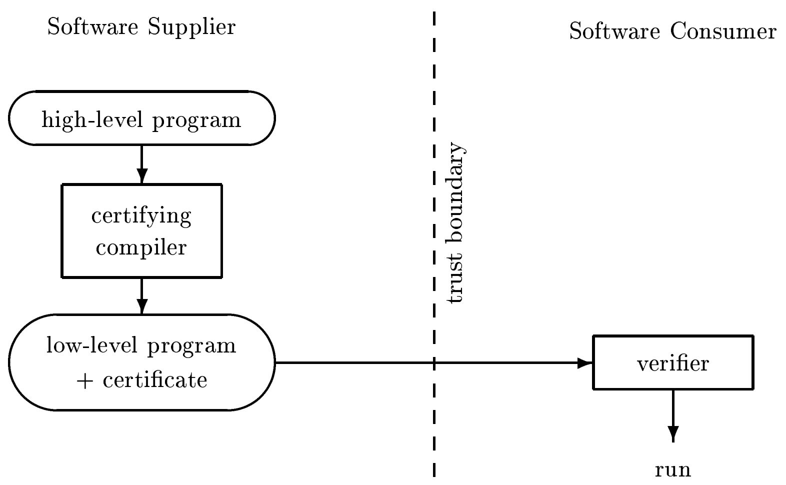

Fig. 1. Language-Based Security (Simplified View)

The key benefit of this approach is that the onus of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier. The supplier must provide a certificate that gives sufficient information to verify that the object code meets the security policy. The consumer's task is thus reduced from the level of *proving* to the level of *checking*, a much simpler matter.

The certificate can take different forms. With PCC, the certificate is a proof in first-order logic of certain verification conditions, and the verification process involves checking that the certificate is indeed a valid first-order proof. With TAL, the certificate is a type annotation, and the verification process involves type checking. With ECC, the certificate is an annotation of the object code that indicates the structure and intention of the code along with some basic type information.

What high-level language constructs best translate to useful information in a certificate? What security policies can be handled? How can we allow consumers to express specialized security policies easily? Can we make certificates concise? How efficiently can the supplier construct them and how efficiently can the consumer verify them? How do we prove that the verification mechanism is correct? By now there are a number of related projects that address these questions and more. Although the various proposals differ in expressiveness, flexibility, and efficiency, they all share a common goal: to use extra information generated during compilation to help make the local execution of untrusted mobile code safe and efficient.

4.1 Java

Perhaps the first large-scale practical instance of the language-based approach was the Java programming language [12]. Java contains a language-based mechanism designed to protect against malicious applets. The Java runtime environment contains a *bytecode verifier* that is supposed to ensure the basic properties of memory, control flow, and type safety. There is also a trusted *security manager* that enforces higher-level safety policies such as restricted disk I/O.

The Java compiler produces platform-independent virtual machine instructions or *bytecode* that can be verified by the consumer before execution. The bytecode is then either interpreted by a Java virtual machine (VM) interpreter or further compiled down to native code.

Early versions of Java contained a number of highly publicized security gaps [4]. For example, one problem was a subtle defect in the Java type system that allowed a partially instantiated class loader to be created under the control of an applet. It was then possible for the applet to use this class loader to load, say, a malicious security manager that would permit unlimited disk access.

According to some authors [4, 13], these problems were ultimately due to a lack of an adequate semantic model for Java. Steps to remedy this situation have since been taken [1, 27]. Nevertheless, despite these initial failings, the basic approach constituted a significant step forward in practical programming language security. It not only pointed the way toward a simple and effective means of providing a basic level of security, but also helped to galvanize the attention of the programming language and verification community on critical security issues engendered by the rise of the Internet.

One disadvantage to the Java system is that the machine-independent bytecode that is produced by the Java compiler is still quite high-level. After downloading, it must either be interpreted by a Java VM interpreter or compiled to native code by a just-in-time (JIT) compiler. Either way, a runtime penalty is incurred. If the safety certificate represented in the bytecode were mapped down to the level of native code by a back-end Java VM compiler, then the same degree of safety could be ensured without the runtime penalty, because the back-end compilation could be done by the code supplier before downloading. This would trade the platform independence of Java VM for the efficiency of native code.

4.2 Proof Carrying Code (PCC)

Proof carrying code (PCC) [21–26] refers to a methodology for allowing formal proofs of general safety properties to be produced and verified before the code is run. The safety conditions are expressed in first-order logic augmented with symbols for various language and machine constructs. The verication process involves a proof generation step on the part of the software supplier and a proof checking step on the part of the software consumer.

The most general version of PCC is somewhat more complicated than indicated in Figure 1, involving a two-phase interaction between the supplier and the consumer. In the first phase of the PCC protocol, the supplier produces a program consisting of annotated object code and sends it to the consumer. The annotation consists of loop invariants and function pre- and postconditions. These annotations make subsequent phases of the protocol easier. The consumer, who has a particular safety policy in mind, generates from the annotated code a *verification condition*, a logical formula that implies that the program satisfies the safety policy, and sends it back to the supplier. A proof of the verification condition constitutes a proof of safety of the program with respect to the consumer's safety policy. The supplier then runs a theorem prover, which produces a proof of the verification condition, then sends the proof back to the consumer. The consumer then runs a proof checker to check that the proof is valid.

The initial annotation of the code is produced by a certifying compiler. The compiler uses information from the program source and program analysis during compilation to construct loop invariants. This process is mostly automatic, but sometimes human intervention is required, depending on the complexity of the security policy. Also included in the initial annotation are pre- and postconditions of functions. The precondition of a function should provide enough information to allow a verification condition to be constructed for that function in the next phase. The verification condition implies that the function satisfies the security policy and satisfies its postcondition.

The Touchstone compiler [22] is a certifying compiler for a type-safe subset of C that implements this phase of PCC. In addition to the object code, it provides information sufficient for constructing a verification condition for type safety. One of the major strengths of the Touchstone compiler is that it admits many common optimizations such as dead code elimination, common subexpression elimination, copy propagation, instruction scheduling, register allocation, loop invariant hoisting, redundant code elimination, and the elimination of array bounds checks.

In the next phase of the PCC protocol, the consumer produces from the code and certificate provided by the code supplier a statement in first-order logic called the *verification condition*. This task is performed by the *verification condition generator* (VCGen). The consumer's security policy is *defined* by the action of the VCGen component; that is, the verification condition that VCGen generates is, by definition, the formal statement of the security policy as instantiated for that particular program. It would be difficult—and for practical purposes, unnecessary—to express the security policy formally independent of any particular program. Part of the definition can be understood in terms of the *action precondition* of each individual operation, which is a formal statement of what it means for that action to be safe locally. However, the safety policy can be more than just the accumulation of all local action preconditions. The verification condition is returned to the software supplier.

The next phase of the protocol involves proving the verification condition. At this point the protocol works entirely in the framework of first-order logic, independent of the original program or programming language. The software supplier constructs a formal proof of the verification condition and returns it to the consumer for checking. The verifier checks that the proof is indeed a valid proof of the verification condition constructed in the previous phase.

In the PCC implementation [22], the verification condition and its proof are encoded using the Edinburgh Logical Framework (LF) [8]. The theorem prover is based on the Nelson-Oppen theorem prover architecture for combined theories. Key tools are congruence closure for dealing with equality and linear simplex for dealing with arithmetic. The latter is important in eliminating array bounds checks.

Necula's PhD thesis [22] describes extensive experiments with PCC giving results on running times and code and proof sizes for various benchmarks.

The advantages of the PCC approach are its expressiveness and the ability to handle code optimizations. In principle, any security policy that can be constructed by a verification condition generator and expressed as a first-order verification condition can be handled. The main disadvantages are that it is a two-phase protocol, that it involves weighty machinery such as a full-fledged first-order theorem prover and proof checker, and that proof sizes are quite large, roughly 2.5 times the size of the object code for type safety and even larger sizes for more complicated safety policies. This makes PCC appropriate for applications requiring fast optimized code that will be verified once but run many times, such as extensions to extensible system kernels, but less attractive for run-once applications such as applets and active messages.

4.3 Typed Assembly Language (TAL)

Typed assembly language (TAL) [7, 13, 14, 16] is a language-based system in which type information from a strongly-typed high-level language is carried down during the compilation process through a series of transformations through a platform-independent typed intermediate language (TIL) [15, 31] and finally down to the level of the object code itself. The result is a type annotation of the object code that can be checked by an ordinary type checker. One can view TAL as a form of proof-carrying code (PCC) in the sense that a complete type annotation is essentially a proof of type safety. In this view, a type checker is essentially a proof checker.

TAL is not as expressive as PCC, but it can handle any security policy that can be expressed in terms of the type system. This includes memory, control flow, and type safety, among others. TAL is also robust with respect to compiler optimizations, since type annotations can be transformed along with the code.

The original version of TAL [16] was rather abstract, compiling down from an polymorphically-typed abstract ML-like language to an idealized RISC-like assembly language. Function call linkages were encoded using continuation passing semantics. This version of TAL already distinguished between initialized and uninitialized data, so that allocation of memory and its initialization need not occur atomically. Deallocation of memory is assumed to be handled by a trusted garbage collector, although some work has been done toward relaxing this assumption [3]. The syntax of the original version of TAL is given in Figure 2.

types	τ	::=	$\alpha \mid \mathrm{int} \mid \forall[\Delta].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau$
initialization flags	φ	::=	$0 \mid 1$
label assignments	Ψ	::=	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$
type assignments	Δ	::=	$\cdot \mid \alpha, \Delta$
register assignments	Γ	::=	$\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$
registers	r	::=	$r_1 \mid \cdots \mid r_k$
word values	w	::=	$\ell \mid i \mid ?\tau \mid w[\tau] \mid \mathrm{pack}\ [\tau, w]\ \mathrm{as}\ \tau'$
small values	v	::=	$r \mid w \mid v[\tau] \mid \mathrm{pack}\ [\tau, v]\ \mathrm{as}\ \tau'$
heap values	h	::=	$\langle w_1, \ldots, w_n \rangle \mid \mathrm{code}[\Delta]\Gamma.I$
heaps	H	::=	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	R	::=	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	i	::=	$\mathrm{aop}\ r_d, r_s, v \mid \mathrm{bop}\ r, v \mid \mathrm{ld}\ r_d, r_s(i) \mid \mathrm{malloc}\ r[\boldsymbol{\tau}]$
			$\mid \mathrm{mov}\ r_d, v \mid \mathrm{st}\ r_d(i), r_s \mid \mathrm{unpack}\ [\alpha, r_d], v$
arithmetic ops	aop	::=	add | sub | mul
branch ops	bop	::=	beq | bneq | bgt | blt | bgte | blte
instruction sequences	I	::=	$\iota; I \mid \mathrm{jmp}\ v \mid \mathrm{halt}\ [\tau]$
programs	P	::=	(H, R, I)

Fig. 2. Syntax of TAL [16]

In order to conform more directly to stack-based runtime architectures, TAL has been extended to include stack types [14]. The syntax of this extension is given in Figure 3.

types	τ	$::=$	$\cdots \mid \mathrm{ns}$
stack types	σ	$::=$	$\rho \mid \mathrm{nil} \mid \tau :: \sigma$
type assignments	Δ	$::=$	$\cdots \mid \rho, \Delta$
register assignments	Γ	$::=$	$\{r_1 : \tau_1, \ldots, r_n : \tau_n, \mathrm{sp} : \sigma\}$
word values	w	$::=$	$\cdots \mid w[\sigma] \mid \mathrm{ns}$
small values	v	$::=$	$\cdots \mid v[\sigma]$
register files	R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathrm{sp} \mapsto S\}$
instructions	i	$::=$	$\cdots \mid \mathrm{salloc}\ n \mid \mathrm{sfree}\ n \mid \mathrm{sld}\ r_d, \mathrm{sp}(i) \mid \mathrm{sst}\ \mathrm{sp}(i), r_s$
stacks	S	$::=$	$\mathrm{nil} \mid w :: S$

Fig. 3. Extension of TAL to accommodate stacks [14]

Other extensions of the TAL approach include type support for modules and static linking [7], eliminating array bounds checks [34], and runtime code generation [9]. A realistic version of TAL for the x86 architecture called TALx86 has been developed, along with a prototype compiler for a type-safe C-like language called Popcorn [13].

4.4 Efficient Code Certification (ECC)

The author's project on efficient code certification (ECC) [11] was conceived as a way to improve the runtime efficiency of small, untrusted, run-once applications such as applets and active messages while still ensuring safe execution. "Run-once" means that the cost of verification cannot be amortized over the lifetime of the code, so certificates should be as concise and easy to verify as possible.

ECC attempts to identify the minimum information necessary to ensure a basic but nontrivial level of code safety, including control flow, memory, and stack safety, and to encapsulate this information in a succinct certificate that is easy to produce and to verify. The level of safety currently provided by the ECC prototype is roughly comparable to that provided by Java bytecode verification; but unlike bytecode, it operates at the level of native code, thus avoiding the runtime overhead of bytecode interpretation or just-in-time compilation. The prototype implementation compiles Scheme to executable x86 machine code.

The system does not rely on general theorem provers or typing mechanisms. Although less flexible than PCC or TAL, certificates are compact and easy to produce and to verify. The certificate can be produced by the code supplier during the code generation phase of compilation and verified by the consumer at load time; both operations are automatic and invisible to both parties.

Although inspired by PCC, it would be inaccurate to call ECC certificates *proofs*, because they are not proofs in any formal system. The certificate consists of annotations that provide information about the structure and intention of the code, as well as some basic typing information. This information is derived from the high-level program during compilation. Guided by this information, the verifier checks a set of simple static conditions that inductively imply the desired safety properties.

Drawbacks to ECC include platform-dependence and fragility with respect to fancy compiler optimizations. Simple local optimizations such as tail recursion elimination can be handled. Preliminary experiments indicate that the sizes of ECC certificates range from 6% to 25% of the size of the object code. This seems to indicate a substantial improvement over PCC, although a fair comparison would require a more careful analysis to take all variables into account.

The main reason for the savings in certificate size in ECC over PCC or TAL is that ECC makes heavy use of compiler conventions. This is both an advantage and a disadvantage. The advantage is that it allows information that must be included explicitly in a PCC or TAL certificate to be omitted from an ECC certificate. The disadvantage is that it makes the verifier heavily dependent on the compiler implementation.

For example, suppose subroutines always return their result in register r. The certificate does not need to say this, but only indicate where the subroutine linkages are. The verifier, knowing about this convention and knowing from the annotation that a certain piece of code is an instance of a standard subroutine linkage, has only to check that the correct subroutine linkage code is there. It may then proceed under the assumption that the result is in register r. Subroutine linkage code generated by the compiler will be fairly uniform—a simple function of the number and type of arguments, modulo work register names—so the check can be done by table lookup with unification on register names. All the certificate has to do is indicate the intention of the code.

The verification process in ECC is very efficient. It is linear time except for a sorting step to sort jump destinations, but since almost all jumps are forward and local, a simple insertion sort suffices.

4.5 Information Flow

Language-based methods can be used to control information flow among mutually distrustful agents [17–19]. This is similar to other forms of safety described in the previous section, except that the security policies are based on a model of information flow. The policy is specified by the user by means of annotations in the high-level language that limit how information can flow in a program and between programs. Annotated programs are then checked at compile time to ensure that they conform to the flow rules.

Currently, a prototype implementation called JFlow has been created that augments Java with information flow primitives [17]. The JFlow compiler is implemented as a source-to-source translator that checks information flow safety using a type-checking mechanism, then discards the annotations, emitting ordinary Java. If the control information were passed down to the object code, then downloaded object code could be verified in a manner similar to TAL or ECC before running to ensure that it does not leak information.

Acknowledgements

I thank Martín Abadi, Jim Ezick, Neal Glew, Peter Lee, Greg Morrisett, Andrew Myers, George Necula, Fred Schneider, and David Walker for valuable conversations. In particular, Neal Glew and Fred Schneider provided extensive comments on an earlier draft. The support of the National Science Foundation under grant CCR-9708915 is gratefully acknowledged.

References

1. M. Abadi and R. Stata. A type system for Java bytecode subroutines. In *Proc. 25th Symp. Principles of Programming Languages*, pages 149–160. ACM SIGPLAN/SIGACT, January 1998.
2. B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. 15th Symp. Operating System Principles*, pages 267–284. ACM, December 1995.
3. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th Symp. Principles of Programming Languages*, pages 262–275. ACM SIGPLAN/SIGACT, January 1999.
4. Drew Dean, Ed Felten, and Dan Wallach. JAVA security: From HotJava to Netscape and beyond. In *Proc. Symp. Security and Privacy*. IEEE, May 1996.
5. Whitfield Diffie and Susan Landau. *Privacy on the Line: The Politics of Wiretapping and Encryption*. MIT Press, 1998.
6. Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective, April 1999. Preprint.
7. N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th Symp. Principles of Programming Languages*, pages 250–261. ACM SIGPLAN/SIGACT, January 1999.
8. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. Assoc. Comput. Mach.*, 40(1):143–184, January 1993.
9. Luke Hornof and Trevor Jim. Certifying compilation and runtime code generation. In *Proc. Workshop on Partial Evaluation and SemanticsBased Program Manipulation*, pages 60–74. ACM, January 1999.
10. Bill Joy and Ken Kennedy, co-chairs. *Information Technology Research: Investing in Our Future*. President's Information Technology Advisory Committee, February 1999. `http://www.ccic.gov/`.
11. Dexter Kozen. Efficient code certification. Technical Report 98-1661, Computer Science Department, Cornell University, January 1998.
12. Tim Lindholm and Frank Yellin. *The JAVA virtual machine specification*. Addison Wesley, 1996.
13. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. Workshop on Compiler Support for System Software*, pages 25–35. ACM SIGPLAN, May 1999.
14. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *Proc. Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer-Verlag, March 1998.

15. G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *1996 Workshop on Compiler Support for Systems Software*, 1996.
16. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN/SIGSIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.
17. Andrew C. Myers. JFlow: Practical static information flow control. In *Proc. 26th Symp. Principles of Programming Languages*. ACM, January 1999.
18. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 16th Symp. Operating System Principles*, pages 129–142. ACM, October 1997.
19. Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. Symp. Security and Privacy*, pages 186–197. IEEE, May 1998.
20. National Coordination Office for Computing, Information, and Communications. *Information Technology for the 21st Century: A Bold Investment in America's Future*, 24 January 1999. Draft. `http://www.ccic.gov/`.
21. George C. Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Programming Languages*, pages 106–119. ACM SIGPLAN/SIGACT, January 1997.
22. George C. Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, September 1998.
23. George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd Symp. Operating System Design and Implementation*. ACM, October 1996.
24. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. Conf. Programming Language Design and Implementation*, pages 333–344. ACM SIGPLAN, 1998.
25. George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proc. 13th Symp. Logic in Computer Science*, pages 93–104. IEEE, June 1998.
26. George C. Necula and Peter Lee. Safe, untrusted agents using using proof-carrying code. In Giovanni Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *Lect. Notes in Computer Science*, pages 61–91. Springer-Verlag, June 1998.
27. Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th Symp. Principles of Programming Languages*, pages 70–78. ACM SIGPLAN/SIGACT, January 1999.
28. Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. 11th Int. Workshop WDAG '97*, volume 1320 of *Lecture Notes in Computer Science*, pages 1–14. ACM SIGPLAN, Springer-Verlag, September 1997.
29. Fred B. Schneider. Enforceable security policies. Technical Report TR98-1664, Computer Science Department, Cornell University, January 1998.
30. Fred B. Schneider, editor. *Trust in Cyberspace*. Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, National Research Council. National Academy Press, 1999.
31. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Conf. Programming Language Design and Implementation*. ACM SIGPLAN, 1996.
32. R. Wahbe, S. Lucco, T. E. Anderson, and S. L Graham. Efficient software-based fault isolation. In *Proc. 14th Symp. Operating System Principles*, pages 203–216. ACM, December 1993.

33. David Walker. A type system for expressive security policies. In *Proc. FLOC'99 Workshop on Run-time Result Verification*, July 1999. To appear.
34. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proc. Conf. Programming Language Design and Implementation*, pages 249–257. ACM SIGPLAN, June 1998.