A Language-Based Approach to Security

Fred B. Schneider[1], Greg Morrisett[1], and Robert Harper[2]

[1] Cornell University, Ithaca, NY
[2] Carnegie Mellon University, Pittsburgh, PA

Abstract. Language-based security leverages program analysis and program rewriting to enforce security policies. The approach promises efficient enforcement of fine-grained access control policies and depends on a trusted computing base of only modest size. This paper surveys progress and prospects for the area, giving overviews of in-lined reference monitors, certifying compilers, and advances in type theory.

1 Introduction

The increasing dependence by industry, government, and society on networked information systems means that successful attacks could soon have widespread and devastating consequences. Integrity and availability now join secrecy as crucial security policies, not just for the military but also for the ever-growing numbers of businesses and individuals that use the Internet. But current systems lack the technology base needed to address these new computer-security needs [16].

For the past few years, we and others have been exploring the extent to which techniques from programming languages—compilers, automated program analysis, type checking, and program rewriting—can help enforce security policies in networked computing systems. This paper explains why this *language-based security* approach is considered so promising, some things that already have been accomplished, and what might be expected. But the paper also can be seen as a testament to research successes in programming languages, giving evidence that the area is poised to have an impact far beyond its traditional scope.

Section 2 discusses two computer security principles that suggest the focus on language-based security is sensible. Then, section 3 discusses the implementation of reference monitors for policy enforcement. Three language-based security paradigms—in-lined reference monitors, type systems, and certifying compilers—are the subject of section 4. Some concluding remarks appear in section 5.

2 Some Classic Principles

Work in language-based security is best understood in terms of two classic computer security principles [15]:

Principle of Least Privilege. Throughout execution, each principal should be accorded the minimum access necessary to accomplish its task.

Minimal Trusted Computing Base. Assurance that an enforcement mechanism behaves as intended is greatest when the mechanism is small and simple.

These principles were first articulated over twenty-five years ago, at a time when economics dictated that computer hardware be shared and, therefore, user computations had to be protected from each other. Since it was kernel-implemented abstractions that were being shared, security policies for isolating user computations were formulated in terms of operating system objects. Moreover, in those days, the operating system kernel itself was small and simple. The kernel thus constituted a minimal trusted computing base that instantiated the Principle of Least Privilege.

Computing systems have changed radically in twenty-five years. Operating system kernels are no longer simple or small. The source code for Windows 2000, for example, comprises millions of lines of code. One reason today's operating systems are so large is to support basic services (*e.g.*, windowing, graphics, distributed file systems) needed for the varied tasks they now perform. But another reason is performance—subsystems are no longer isolated from each other to avoid expensive context switches during execution. For example, the graphics subsystem of Windows is largely contained within the kernel's address space(!) to reduce the cost of invoking common drawing routines. So operating system kernels today constitute an unmanageably large and complicated computing base—a far cry from the minimal trusted computing base we seek.

Moreover, today's operating system kernels enforce only coarse-grained policies.

- Almost all code for a given machine is run on behalf of a single user, and principals are equated with users. Consequently, virtually all code runs as a single principal under a single policy (*i.e.*, a single set of access permissions).
- Many resources are not implemented by the operating systems kernel. Thus, the kernel is unable to enforce the policies needed for protecting most of the system's resources.

This *status quo* allows viruses, such as Melissa and the Love Bug, to propagate by hiding within an email message a script that is transparently invoked by the mail-viewer application (without an opportunity for the kernel to intercede) when the message is opened.[1] In short, today's operating systems do not and cannot enforce policies concerning application-implemented resources, and individual subsystems lack the clear boundaries that would enable policies concerning the resources they manage to be enforced.

[1] Because the script runs with all the privileges of the user that received the message, the virus is able to read the user's address book and forward copies of itself, masquerading as personal mail from a trusted friend.

Though ignored today, Principle of Least Privilege and Minimal Trusted Computing Base, remain sound and sensible principles, as they are independent of system architecture, computer speed, and the other dimensions of computer systems that have undergone radical change. Traditional operating system instantiations of these principles might no longer be feasible, but that does not preclude using other approaches to policy enforcement. Language-based security is one such approach.

3 The Case for Language-Based Security

A *reference monitor* observes execution of a target system and halts that system whenever it is about to violate some security policy of concern. Security mechanisms found in hardware and system software typically either implement reference monitors directly or are intended to facilitate the implementation of reference monitors. For example, an operating system might mediate access to files and other abstractions it supports, thereby implementing a reference monitor for policies concerning those objects. As another example, the context switch (trap) caused whenever a system call instruction is executed forces a transfer of control, thereby facilitating invocation of a reference monitor whenever a system call is executed.

To do its job, a reference monitor must be protected from subversion by the target systems it monitors. Memory protection hardware, which ensures that execution by one program cannot corrupt the instructions or data of another, is commonly used for this purpose. But placing the reference monitor and target systems in separate address spaces has a performance cost and restricts what policies can be enforced.

- The performance cost results from the overhead due to context switches associated with transferring control to the reference monitor from within the target systems. The reference monitor must receive control whenever a target system participates in an event relevant to the security policy being enforced. In addition, data must be copied between address spaces.
- The restrictions on what policies can be enforced arise from the means by which target system events cause the reference monitor to be invoked, since this restricts the vocabulary of events that can be involved in security policies. Security policies that govern operating system calls, for example, are feasible because traps accompany systems calls.

The power of the Principle of Least Privilege depends on having flexible and general notions of principal and minimum access. Any interface—not just the user/kernel interface—might define the objects governed by a security policy. And an expressive notion of principal is needed if enforcement decisions might depend on, among other things, the current state of the machine, past execution history, who authored the code, on who's behalf is the code executing, and so on.

Language-based security, being based on program analysis and program rewriting, supports the flexible and general notions of principal and minimum access needed in order the instantiate the Principle of Least Privilege. In particular, software, being universal, can always provide the same functionality (if not performance) as a reference monitor. An interpreter, for instance, could include not only the same checks as found in hardware or kernel-based protection mechanisms but also could implement additional checks involving the application's current and past states.

The only question, then, is one of performance. If the overhead of unadulterated interpretation is too great, then compilation technology, such as just-in-time compilers, partial evaluation, run-time code generation, and profile-driven feedback optimization, can be brought to bear. Moreover, program analysis, including type-checking, dataflow analysis, abstract interpretation, and proof-checking, can be used to reason statically about the run-time behavior of code and eliminate unnecessary run-time policy enforcement checks.

Beyond supporting functionality equivalent to hardware and kernel-supported reference monitoring, the language-based approach to security offers other benefits. First, language-based security yields policy enforcement solutions that can be easily extended or changed to meet new, application-specific demands. Second, if a high-level language (such as Java or ML) is the starting point, then linguistic structures, such as modules, abstract data types, and classes, allow programmers to specify and encapsulate application-specific abstractions. These same structures can then provide a vocabulary for formulating fine-grained security policies. Language-based security is not, however, restricted to systems that have been programmed in high-level languages. In fact, much work is directed at enforcing policies on object code because (i) the trusted computing base is smaller without a compiler and (ii) policies can then be enforced on programs for which no source code is available.

EM Security Policies

A program analyzer operating on program text (source or object code) has more information available about how that program could behave than does a reference monitor observing a single execution.[2] This is because the program text is a terse representation of all possible behaviors and, therefore, contains information—about alternatives and the future—not available in any single execution. It would thus seem that, ignoring questions of decidability, program analysis can enforce policies that reference monitors cannot. To make the relationship precise, the class of security policies that reference monitors can enforce was characterized in [17], as follows.

A *security policy* defines execution that, for one reason or another, has been deemed unacceptable. Let EM (for Execution Monitoring) be the class of security policies that can be enforced by monitoring execution of a target system and

[2] We are assuming that a reference monitor sees only security-relevant actions and values. Once the entire state of the system becomes available, then the reference monitor would have access to the program text.

terminating execution that is about to violate the security policy being enforced. Clearly, EM includes those policies that can be enforced by security kernels, reference monitors, firewalls, and most other operating system and hardware-based enforcement mechanisms that have appeared in the literature. Target systems may be objects, modules, processes, subsystems, or entire systems; the execution steps monitored may range from fine-grained actions (such as memory accesses) to higher-level operations (such as method calls) to operations that change the security-configuration and thus restrict subsequent execution.

Mechanisms that use more information than would be available only from monitoring a target system's execution are, by definition, excluded from EM. Information provided to an EM mechanism is thus insufficient for predicting future steps the target system might take, alternative possible executions, or all possible target system executions. Therefore, compilers and theorem-provers, which analyze a static representation of a target system to deduce information about all of its possible executions, are not considered EM mechanisms. Also excluded from EM are mechanisms that modify a target system before executing it. The modified target system would have to be "equivalent" to the original (except for aborting executions that would violate the security policy of interest), so a definition for "equivalent" is thus required to analyze this class of mechanisms.

We represent target system executions by finite and infinite sequences, where Ψ denotes a universe of all possible finite and infinite sequences. The manner in which executions are represented is irrelevant here. Finite and infinite sequences of atomic actions, of higher-level system steps, of program states, or of state/action pairs are all plausible alternatives. A target system S defines a subset Σ_S of Ψ corresponding to the executions of S.

A characterization of EM-enforceable security policies is interesting only if the definition being used for "security policy" is broad enough so that it does not exclude things usually considered security policies.[3] Also, the definition must be independent of how EM is defined, for otherwise the characterization of EM-enforceable security policies would be a tautology, hence uninteresting. We therefore adopt the following.

Definition of Security Policy: A *security policy* is specified by giving a predicate on sets of executions. A target system S *satisfies* security policy $\mathcal{P}$ if and only if $\mathcal{P}(\Sigma_S)$ equals *true*.

By definition, enforcement mechanisms in EM work by monitoring execution of the target. Thus, any security policy $\mathcal{P}$ that can be enforced using a mechanism from EM must be specified by a predicate of the form

$$\mathcal{P}(\Pi): \ (\forall \sigma \in \Pi: \ \widehat{\mathcal{P}}(\sigma)) \tag{1}$$

where $\widehat{\mathcal{P}}$ is a predicate on (individual) executions. $\widehat{\mathcal{P}}$ formalizes the criteria used by the enforcement mechanism for deciding to terminate an execution that would otherwise violate the policy being enforced. In [1] and the literature on linear-time concurrent program verification, a set of executions is called a *property* if

[3] However, there is no harm in being liberal about what is considered a security policy.

set membership is determined by each element alone and not by other members of the set. Using that terminology, we conclude from (1) that a security policy must be a property in order for that policy to have an enforcement mechanism in EM.

Not every security policy is a property. Some security policies cannot be defined using criteria that individual executions must each satisfy in isolation. For example, information flow policies often characterize sets that are not properties (as proved in [10]). Whether information flows from variable x to y in a given execution depends, in part, on what values y takes in other possible executions (and whether those values are correlated with the value of x). A predicate to specify such sets of executions cannot be constructed only using predicates defined on single executions in isolation.

Not every property is EM-enforceable. Enforcement mechanisms in EM cannot base decisions on possible future execution, since that information is, by definition, not available to such a mechanism. Consider security policy $\mathcal{P}$ of (1), and suppose σ' is the prefix of some finite or infinite execution σ where $\widehat{\mathcal{P}}(\sigma) = \mathit{true}$ and $\widehat{\mathcal{P}}(\sigma') = \mathit{false}$ hold. Because execution of a target system might terminate before σ' is extended into σ, an enforcement mechanism for $\mathcal{P}$ must prohibit σ' (even though supersequence σ satisfies $\widehat{\mathcal{P}}$).

We can formalize this requirement as follows. For σ a finite or infinite execution having i or more steps, and τ' a finite execution, let

$\sigma[..\mathrm{i}]$ denote the prefix of σ involving its first i steps
$\tau'\,\sigma$ denote execution τ' followed by execution σ

and define Π^- to be the set of all finite prefixes of elements in set Π of finite and/or infinite sequences. Then, the above requirement for $\mathcal{P}$—that $\mathcal{P}$ is *prefix closed*—is:

$$(\forall \tau' \in \Psi^-\colon\ \neg\widehat{\mathcal{P}}(\tau') \Rightarrow (\forall \sigma \in \Psi\colon\ \neg\widehat{\mathcal{P}}(\tau'\sigma))) \tag{2}$$

Finally, note that any execution rejected by an enforcement mechanism must be rejected after a finite period. This is formalized by:

$$(\forall \sigma \in \Psi : \neg\widehat{\mathcal{P}}(\sigma) \Rightarrow (\exists i : \neg\widehat{\mathcal{P}}(\sigma[..i]))) \tag{3}$$

Security policies satisfying (1), (2), and (3) are *safety properties* [8], properties stipulating that no "bad thing" happens during any execution. Formally, a property Γ is defined in [9] to be a safety property if and only if, for any finite or infinite execution σ,

$$\sigma \notin \Gamma \Rightarrow (\exists i\colon\ (\forall \tau \in \Psi\colon\ \sigma[..i]\,\tau \notin \Gamma)) \tag{4}$$

holds. This means that Γ is a safety property if and only if Γ can be characterized using a set of finite executions that are prefixes of all executions excluded from Γ. Clearly, a security policy $\mathcal{P}$ satisfying (1), (2), and (3) has such a set of finite prefixes—the set of prefixes $\tau' \in \Psi^-$ such that $\neg\widehat{\mathcal{P}}(\tau')$ holds—so $\mathcal{P}$ is satisfied by sets that are safety properties according to (4).

The above analysis of enforcement mechanisms in EM has established:

Non EM-Enforceable Security Policies: If the set of executions for a security policy $\mathcal{P}$ is not a safety property, then an enforcement mechanism from EM does not exist for $\mathcal{P}$.

One consequence is that ruling-out additional executions never causes an EM-enforceable policy to be violated, since ruling-out executions never invalidates a safety property. Thus, an EM enforcement mechanism for any security policy $\mathcal{P}'$ satisfying $\mathcal{P}' \Rightarrow \mathcal{P}$ also enforces security policy $\mathcal{P}$. However, a stronger policy $\mathcal{P}'$ might proscribe executions that do not violate $\mathcal{P}$, so using $\mathcal{P}'$ is not without potentially significant adverse consequences. The limit case, where $\mathcal{P}'$ is satisfied only by the empty set, illustrates this problem.

Second, Non EM-Enforceable Security Policies implies that EM mechanisms compose in a natural way. When multiple EM mechanisms are used in tandem, the policy enforced by the aggregate is the conjunction of the policies that are enforced by each mechanism in isolation. This is attractive, because it enables complex policies to be decomposed into conjuncts, with a separate mechanism used to enforce each of the component policies.

We can use the Non EM-Enforceable Security Policies result to see whether or not a given security policy might be enforced using a reference monitor (or some other form of execution monitoring). For example, access control policies, which restrict what operations principals can perform on objects, define safety properties. (The set of proscribed partial executions contains those partial executions ending with an unacceptable operation being attempted.) Information flow policies do not define sets that are properties (as discussed above). And, availability policies, if taken to mean that no principal is forever denied use of some given resource, is not a safety property—any partial execution can be extended in a way that allows a principal to access the resource, so the defining set of proscribed partial executions that every safety property must have is absent. Thus we conclude that access control policies can be enforced by reference monitors but neither information flow nor availability policies (as we formulated them) can be.

4 Enforcing Security Policies

The building blocks of language-based security are program rewriting and program analysis. By rewriting a program, we can ensure that the result is incapable of exhibiting behavior disallowed by some security policy at hand. And by analyzing a program, we ensure only those programs that cannot violate the policy are ever given an opportunity to be executed.

That is the theory. Actual embodiments of the language-based security vision invariably combine program rewriting and program analysis. Today's research efforts can be grouped into two schools. One—in-lined reference monitors—takes program rewriting as a starting point; the other—type-safe programming languages—takes program analysis, as a starting point. In what follows, we discuss the strengths and weaknesses of each of these schools. We then discuss an emerging approach—certifying compilation—and how the combination of all three techniques (rewriting, analysis, and certification) yield a comprehensive security framework.

4.1 In-lined Reference Monitors

An alternative to placing the reference monitor and the target system in separate address spaces is to modify the target system code, effectively merging the reference monitor in-line. This is, in effect, what is done by software-fault isolation (SFI), which enforces the security policy that prevents reads, writes, or branches to memory locations outside of certain predefined memory regions associated with a target system [20]. But a reference monitor for any EM security policy could be merged into a target application, provided the target can be prevented from circumventing the merged code.

Specifying such an *in-lined reference monitor* (IRM) involves defining [6]

- *security events*, the policy-relevant operations that must be mediated by the reference monitor;
- *security state*, information stored about earlier security events that is used to determine which security events can be allowed to proceed; and
- *security updates*, program fragments that are executed in response to security events and that update the security state, signal security violations, and/or take other remedial action (*e.g.*, block execution).

A load-time, trusted *IRM rewriter* merges checking code into the application itself, using program analysis and program rewriting to protect the integrity of those checks. The IRM rewriter thus produces a *secured application*, which is guaranteed not to take steps violating the security policy being enforced. Notice, with the IRM approach, the conjunction of two policies can be enforced by passing the target application through the IRM rewriter twice in succession—once for each policy. And also, by keeping policy separate from program, the approach makes it easier to reason about and evolve the security of a system.

Experiments with two generations of IRM enforcement suggest that the approach is quite promising. SASI (Security Automata SFI Implementation), the first generation, comprised two realizations [5]. One transformed Intel x86 assembly language; the other transformed Java Virtual Machine Language (JVML). Second generation IRM enforcement tools PoET/PSLang, (Policy Enforcement Toolkit/Policy Specification Language) transformed JVML [6].

The x86 SASI prototype works with assembly language output of the GNU `gcc` C compiler. Object code produced by `gcc` observes certain register-usage conventions, is not self-modifying, and is guaranteed to satisfy two assumptions:

- Program behavior is insensitive to adding stutter-steps (*e.g.*, nop's).
- Variables and branch-targets are restricted to the set of labels identified by `gcc` during compilation.

These restrictions considerably simplify the task of preventing code for checking and for security updates from being corrupted by the target system. In particular, it suffices to apply x86 SASI with the simple memory-protection policy enforced by SFI in order to obtain target-system object code that cannot subvert merged-in security state or security updates.

The JVML SASI prototype exploits the type safety of JVML programs to prevent merged-in variables and state from being corrupted by the target system in which it resides. In particular, variables that JVML SASI adds to a JVML object program are inaccessible to that program by virtue of their names and types; and code that JVML SASI adds cannot be circumvented because JVML type-safety prevents jumps to unlabeled instructions—these code fragments are constructed so they do not contain labels.[4]

The type-safety of JVML also empowers the JVML SASI user who is formulating a security policy that concerns application abstractions. JVML instructions contain information about classes, objects, methods, threads, and types. This information is made available (though platform-specific functions) to the author of a security policy. Security policies for JVML SASI thus can define permissible computations in terms of these application abstractions. In contrast, x86 code will contain virtually no information about a C program it represents, so the author of a security policy for x86 SASI may be forced to synthesize application events from sequences of assembly language instructions.

Experience with the SASI prototypes has proved quite instructive. A reference monitor that checks every machine language instruction initially seemed like a powerful basis for defining application-specific security policies. But we learned from SASI that, in practice, this power is difficult to harness. Most x86 object code, for example, does not make explicit the application abstractions that are being manipulated by that code. There is no explicit notion of a "function" in x86 assembly language, and "function calls" are found by searching for code sequences resembling the target system's calling convention. The author of a security policy thus finds it necessary to embed a disassembler (or event synthesizer) within a security policy description. This is awkward and error-prone.

One solution would be to obtain IRM enforcement by rewriting high-level language programs rather than object code. Security updates could be merged into the high-level language program (say) for the target system rather than being merged into the object code produced by a compiler. But this is unattractive because an IRM rewriter that modifies high-level language programs adds a compiler to the trusted computing base. The approach taken in JVML SASI seemed the more promising, and it (along with a desire for a friendlier language for policy specification) was the motivation for PoET/PSLang. The lesson is to rely on annotations of the object code that are easily checked and that expose application abstractions. And that approach is not limited to JVML code or

[4] JVML SASI security policies must also rule out indirect ways of compromising the variables or circumventing the code added for policy enforcement. For example, JVML's dynamic class loading and program reflection must be disallowed.

even to type-safe high-level languages. Object code for x86 could include the necessary annotations by using TAL [11] (discussed below).

4.2 Type Systems

Type-safe programming languages, such as ML, Modula, Scheme, or Java, ensure that operations are only applied to appropriate values. They do so by guaranteeing a number of inter-related safety properties, including *memory safety* (programs can only access appropriate memory locations) and *control safety* (programs can only transfer control to appropriate program points).

Type systems that support type abstraction then allow programmers to specify new, abstract types along with signatures for operations that prevent unauthorized code from applying the wrong operations to the wrong values. For example, even if we represent file descriptors as integers, we can use type abstraction to ensure that only integers created by our implementation are passed to file-descriptor routines. In this respect, type systems, like IRMs, can be used to enforce a wider class of fine-grained, application-specific access policies than operating systems. In addition, abstract type signatures provide the means to enrich the vocabulary of an enforcement mechanism in an application-specific way.

The key idea underlying the use of type systems to enforce security policies is to shift the burden of proving that a program complies with a policy from the code recipient (the end user) to the code producer (the programmer). Not only are familiar run-time mechanisms (*e.g.*, address space isolation) insufficiently expressive for enforcing fine-grained security policies but, to the extent that they work at all, these mechanisms impose the burden of enforcement on the end user through the imposition of dynamic checks. In contrast, type-based methods impose on the programmer the burden of demonstrating compliance with a given security policy. The programmer must write the program in conformance with the type system; the end user need only type check the code to ensure that it is safe to execute.

The only run-time checks required in a type-based framework are those necessary for ensuring soundness of the type system itself. For example, the type systems of most commonly-used programming languages do not attempt to enforce value-range restrictions, such as the requirement that the index into an array is within bounds. Instead, any integer-valued index is deemed acceptable but a run-time check is imposed to ensure that memory safety is preserved.

However, it is important to note that the need to dynamically check values, such as array indices, is not inherent to type systems. Rather, the logics underlying today's type systems are too weak, so programmers are unable to express the conditions necessary to ensure soundness statically. This is largely a matter of convenience, though. It is possible to construct arbitrarily expressive type systems with the power of any logic. Such type systems generally require sophisticated theorem provers and programmer guidance in the construction of a proof of type soundness. For example, recent work on dependent type systems [3, 21] extends type checking to include the expression of value-range restrictions sufficient to ensure that array bounds checks may (in many cases) be eliminated, but programmers must add additional typing annotations (*e.g.*, loop invariants) to aid the type checker.

Fundamentally, the only limitation on the expressiveness of a type system is the effort one is willing to expend demonstrating type correctness. Keep in mind that this is a matter of proof—the programmer must demonstrate to the checker that the program complies with the safety requirements of the type system. In practice, it is common to restrict attention to type systems for which checking is computable with a reasonable complexity bound, but more advanced programming systems such as NuPRL [3] impose no such restrictions and admit arbitrary theorem proving for demonstrating type safety.

In summary, advances in the design of type systems now make it possible to express useful security properties and to enforce them in a lightweight fashion, all the while minimizing the burden on the end user to enforce memory and control safety.

4.3 Certifying Compilers

Until recently, the primary weakness of type-based approaches to ensuring safety has been that they relied on

High-Level Language Assumption. The program must be written in a programming language having a well-defined type system and operational semantics.

In particular, the programmer is obliged to write code in the high-level language, and the end user is obliged to correctly implement both its type system (so that programs can be type checked) and its operational semantics (so that it can be executed). These consequences would have questionable utility if they substantially increased the size of the trusted computing base or they reduced the flexibility with which systems could be implemented. But they don't have to. Recent developments in compiler technology are rapidly obviating the High-Level Language Assumption without sacrificing the advantages of type-based approaches. We now turn to that work.

A *certifying compiler* is a compiler that, when given source code satisfying a particular security policy, not only produces object code but also produces a *certificate*—machine-checkable evidence that the object code respects the policy. For example, Sun's `javac` compiler takes Java source code that satisfies a type-safety policy, and it produces JVML code that respects type-safety. In this case, the "certificate" is the type information embedded within the JVML bytecodes.

Certifying compilers are an important tool for policy enforcement because they do their job from outside the trusted computing base. To verify that the output object code of a certifying compiler respects some policy, an automated *certificate checker* (that is part of the trusted computing base) is employed. The certificate checker analyzes the output of the certifying compiler and verifies that this object code is consistent with the characterization given in a certificate.

For example, a JVML bytecode verifier can ensure that bytecodes are type-safe independent of the Java compiler that produced them.

Replacing a trusted compiler with an untrusted certifying compiler plus a trusted certificate checker is advantageous because a certificate checker, including type-checkers or proof-checkers, is typically much smaller and simpler than a program that performs the analysis and transformations needed to generate certified code. Thus, the resulting architecture has a smaller trusted computing base than would an architecture that employed a trusted compiler or analysis.

Java is perhaps the most widely disseminated example of this certifying compiler architecture. But the policy supported by the Java architecture is restricted to a relatively primitive form of type-safety, and the bytecode language is still high-level, requiring either an interpreter or just-in-time compiler for execution.

The general approach of certifying compilation is really quite versatile. For instance, building on the earlier work of the TIL compiler [19], Morrisett *et al.* showed that it is possible to systematically build type-based, certifying compilers for high-level languages that produce Typed Assembly Language (TAL) for concrete machines (as opposed to virtual machines) [12]. Furthermore, the type system of TAL supports some of the refinements, such as value ranges, needed to avoid the overhead of dynamic checks. Nonetheless, as it stands today, the set of security policies that TAL can enforce are essentially those that can be realized through traditional notions of type-safety.

Perhaps the most aggressive instance of certifying compilers was developed by Necula and Lee, who were the first to move beyond implicit typing annotations and develop an architecture in which certificates were explicit. The result, called Proof-Carrying Code (PCC) [13, 14], enjoys a number of advantages over previous work. In particular, the axioms, inference rules, and proofs of PCC are represented as terms in a meta-logical programming language called LF [7], and certificate checking corresponds to LF type-checking. The advantages of using a meta-logical language are twofold:

– It is relatively simple to customize the logic by adding new axioms or inference rules.

– Meta-logical type checkers can be quite small, so in principle a PCC-based system can have an extremely small trusted computing base. For example, Necula implemented an LF type checker that is about 6 pages of C code [13].

Finally, unlike the JVML or TAL, PCC is not limited to enforcing traditional notions of type safety. It is also not limited to EM policies. Rather, as long as the logic is expressive enough to state the desired policy, and as long as the certifying compiler can construct a proof in that logic that the code will respect the policy, then a PCC-based system can check conformance.

4.4 Putting the Technologies Together

Combine in-lined reference monitors, type systems, and certifying compilers—the key approaches to language-based security—and the sum will be greater than the parts. In what follows, we discuss the remarkable synergies among these approaches.

Integrating IRM enforcement with Type Systems. Static type systems are particularly well-suited for enforcing security policies that have been negotiated in advance. Furthermore, enforcement through static checking usually involves less overhead than a more dynamic approach. And finally, static type systems hold the promise of enforcing liveness properties (*e.g.*, termination) and polices that are not properties (*e.g.*, absence of information flow)—things that reference monitors cannot enforce. However, static type systems are ill-suited for the enforcement of policies that depend upon things that can be detected at runtime but cannot be ascertained during program development. Also, it may be simpler to insert a dynamic check than to have a programmer develop an explicit proof that the check is not needed. Consequently, by combining IRMs with advanced type systems, we have both the opportunity to enforce a wider class of policies and more flexibility in choosing an appropriate enforcement mechanism.

Extending IRM enforcement with Certifying Compilers. Program rewriting without subsequent optimization generally leads to systems exhibiting poor performance. However, an IRM rewriter could reduce the performance impact of added checking code by inserting checks only where there is some chance that a security update actually needs to be performed. For example, in enforcing a policy that stipulates messages are never sent after certain files have been read, an IRM rewriter needn't insert code before and after every instruction. A small amount of simple analysis would allow insertions to be limited to those instructions involving file reads and message sends; and a global analysis might allow more aggressive optimizations. Optimization technology, then, can recover performance for the IRM approach.

But an IRM rewriter that contains a global optimizer is larger and more complicated than one that does not. Any optimizations had better always be done correctly, too, since bugs might make it possible for the security policy at hand to be violated. So, optimization routines—just like the rest of the IRM rewriter—are part of the trusted computing base. In the interest of keeping the trusted computing base small, we should hesitate to employ a complicated IRM rewriter.

Must an IRM architecture sacrifice performance on the alter of minimizing the trusted computing base? Not if the analysis and optimization are done with the lesson of certifying compilers in mind. An IRM rewriter can add checking code and security updates and then do analysis and optimization to remove unnecessary checking code, provided the IRM rewriter produces a certificate along with the modified object code. That certificate should describe what code was added everywhere and the analysis that allowed code to be deleted, thereby enabling a certificate checker (in the trusted computing base) to establish independently that the output of the IRM rewriter will indeed never violate the security policy of interest. Thus, the IRM rewriter is extended if ideas from certifying compilers are adopted.

Extending Certifying Compilers with IRM enforcement. Certifying compilers are limited to analysis that can be done automatically. And, unfortunately, there are deep mathematical reasons why certain program analysis cannot be automated—analysis that would be necessary for policies much simpler than found in class EM. Must a certifying compiler architecture sacrifice expressiveness on the alter of automation?

In theory, it would seem so. But in practice, much analysis becomes possible when program rewriting is first allowed. This is an instance of the familiar tradeoff between static and dynamic checks during type checking. For instance, rather than verifying at compile time that a given array index never goes out of bounds, it is a simple matter to have the compiler emit a run-time check. Static analysis of the modified program is guaranteed to establish that the array access is never out of bounds (because the added check prevents it from being so).

The power of a certifying compilers is thus amplified by the capacity to do program rewriting. In the limit, what is needed is the means to modify a program and obtain one in which a given security policy is not violated—exactly what an IRM rewriter does. Thus, the power of certifying compilers is extended if deployed in concert with an IRM rewriter.

5 Concluding Remarks

In-lined reference monitors, certifying compilers, and advanced type systems are promising approaches to system security. Each allows rich instantiations of the Principle of Least Privilege; each depends on only a minimal trusted computing base, despite the ever-growing sizes for today's operating systems, compilers, and programming environments.

The idea of using languages and compilers to help enforce security policies is not new. The Burroughs B-5000 system required applications to be written in a high-level language (Algol), and the Berkeley SDS-940 system employed object-code rewriting as part of its system profiler. More recently, the SPIN [2], Vino [22, 18], and Exokernel [4] extensible operating systems have relied on language technology to protect a base system from a limited set of attacks by extensions.

What is new in so-called language-based security enforcement is the degree to which language semantics provides the leverage. The goal is to obtain integrated mechanisms that work for both high-level and low-level languages; that are applicable to an extremely broad class of fine-grained security policies; and that allow flexible allocation of work and trust among the elements responsible for enforcement.

Acknowledgments

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Schneider is supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and

Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation.

Morrisett is supported in part by AFOSR grant F49620-00-1-0198, and the National Science Foundation under Grant No. EIA 97-03470.

Harper is 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.

References

1. B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters* 21(4):181–185, Oct. 1985.
2. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM Symp. on Operating System Principles (SOSP)*, pages 267–284, Copper Mountain, Dec. 1995.
3. R. L. Constable *et al. Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, 1986.
4. D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Operating System Principles (SOSP)*, Copper Mountain, 1995.
5. U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, Ontario, Canada, Sept. 1999.
6. U. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.
7. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.
8. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
9. L. Lamport. Logical Foundation. In *Distributed Systems-Methods and Tools for Specification*, pages 119-130, Lecture Notes in Computer Science, Vol 190. M. Paul and H.J. Siegert, editors. Springer-Verlag, 1985, New York.
10. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 79–93, Oakland, Calif., May 1994.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 85–97, San Diego California, USA, January 1998.
12. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
13. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.
14. G. C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, Jan. 1997.

15. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 9(63), Sept. 1975.
16. F. B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, Washington, D.C., 1999.
17. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2(4), Mar. 2000.
18. M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, Washington, Oct. 1996.
19. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conf. on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.
20. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 203–216, Asheville, Dec. 1993.
21. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257, Montreal Canada, June 1998.
22. E. Yasuhiro, J. Gwertzman, M. Seltzer, C. Small, K. A. Smith, and D. Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard Computer Center for Research in Computing Technology, 1994.