# Language-Based Security for Malicious Mobile Code

Fred B. Schneider
fbs@cs.cornell.edu

Dexter Kozen
kozen@cs.cornell.edu

Greg Morrisett
greg@eecs.harvard.edu

Andrew C. Myers
andru@cs.cornell.edu

## 1 Introduction

The need for secure computing first became apparent in the early 1970's, when the high cost of hardware forced users to share standalone computers by time-multiplexing the processor. Concurrent processes had to be isolated from each other in order to prevent the bugs of one process from disrupting the execution of another. Different processes resided in separate regions of memory and used the processor during disjoint time intervals. Security policies governed access to shared resources, such as long-term storage, so that they could be shared in a safe and controlled way. The operating system, through a hardware-based *reference monitor*, enforced these policies by intercepting critical operations and blocking them if necessary. Reference monitors and mechanisms enforcing program isolation were small and understandable, so one could be confident that these security policies would be enforced.

Today's operating systems are much more complex than those of the 1970's. They consist of tens of millions of lines of code and must be frequently updated with patches, new device drivers, and other modules received over the Internet. The extensions might come from authenticated sources, but it is difficult to design code that works correctly in every context. There is little basis for confidence in large pieces of code that evolve rapidly with frequent updates. So it is no surprise that with alarming regularity, attackers exploit logic and coding errors to compromise confidentiality, integrity, or availability of computing systems.

Application software that runs above the operating system is also frequently updated by extensions, often packaged as applets or scripts. Content received over the Internet, such as email messages and web pages, typically includes active elements that are executed upon receipt, often without a user's knowledge or ability to intervene. As a result, users today are susceptible to a new class of attacks—attacks not against the operating system but against applications. Since applications run with the privileges of a user, these attacks have access to confidential data and can easily compromise the integrity of the host and data. Virus scanners have proven effective for blocking some known attacks of this form, but they cannot detect previously unseen or obfuscated threats.

In short, today's computers run a collection of software that is dynamically changing, and users often cannot determine what software is actually being executed, what caused it to execute,

or what it might do. Classical operating system mechanisms for isolation and reference monitors provide only limited protection, because they do not distinguish between legitimate application code running on behalf of the user and code provided by an attacker. Furthermore, system software has grown too large and complicated to be considered trustworthy. New technology is needed to build systems that are secure and that we can trust.

# 2 Language-Based Security

The efforts of this project focus on rich classes of security policies and corresponding enforcement mechanisms that can be used to establish trust in system and application software, regardless of origin. Such mechanisms must remain effective despite the ever-increasing size of operating systems and the constant changes that system and application software must undergo. The enabling technologies come from research in *programming languages*, including semantics, type systems, program logics, compilers, and runtime systems. We believe that research in these areas is capable of delivering the flexible, general, and high-performance enforcement mechanisms we seek as well as providing a basis for assurance in the resulting enforcement mechanisms. Here is our rationale.

- The functionality of any hardware-based mechanism can always be achieved by software alone, since one can build an interpreter that does the same checks as the hardware. If the overhead of interpretation is too great, the performance gap can be closed by using compilation technologies, such as just-in-time compilers, partial evaluation, runtime code generation, and profile-driven feedback optimization. Furthermore, unlike hardware realizations, software implementations can be more easily extended or changed to meet new, application-specific demands as the context changes. Indeed, in Section 3, we describe an approach that allows application-specific policies to be easily specified and dynamically composed.

- Modern high-level languages, including Java and ML, provide linguistic structure such as modules, abstract data types, and classes that allow programmers to specify and encapsulate application-specific abstractions. The interfaces of these abstractions can then be used to enrich the vocabulary of the security framework with application-specific concepts, principals, and operations.

- Code analysis, including type checking, dataflow analysis, abstract interpretation, and proof checking, can be leveraged to reason statically about the runtime behavior of code. This allows us to enforce policies, such as restrictions on information flow, that are impossible to implement using runtime mechanisms such as reference monitors. Program analysis also allows further code optimization to eliminate unnecessary dynamic checks. The program analyzer, which could well be a large and complex piece of code, can be replaced in a trusted computing base (TCB) by a proof checker, which will typically be much smaller.

In short, by analyzing code before it executes, runtime checks are avoided, a wider class of policies can be enforced, and the trusted computing base can be relocated. By modifying code before it executes, policies that involve application-specific abstractions and arbitrary program

interfaces can be enforced. Finally, by using high-level language abstractions, we can effectively enrich the vocabulary of the security policy and its enforcement mechanisms.

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 [5], Vino [52, 47], and Exokernel [13] 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 our work is the degree to which language semantics provides the leverage. We have examined integrated mechanisms that work for both high- 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.

In the following, we report on four specific focus areas where we have successfully applied language technology to increase the system security: in-lined reference monitors for application-level safety properties (Section 3), type systems for ensuring safety of low-level legacy languages (Section 4), type systems for end-to-end confidentiality and integrity (Section 5), and secure languages for firmware (Section 6).

# 3 Inlined Reference Monitors

SFI (software fault isolation) employs program modification to rewrite binaries so that only reads, writes, or branches to appropriate locations in a program's address space are allowed [51]. This *memory safety* security policy is useful for protecting a base system (e.g., a kernel) from certain misbehavior by extensions as well as for protecting different users' programs from each other. Though traditionally enforced by address translation hardware, supporting memory safety by program modification has two advantages: (i) it reduces the overhead of cross-domain procedure calls and (ii) it can implement a more-flexible memory-safety model.

SFI-inserted code can be viewed as a reference monitor that has been "in-lined" into a *target application*. This observation led us to investigate the fundamental limits and engineering issues associated with using in-lined reference monitors (IRMs) in general. With the IRM approach, a security policy is specified in a declarative language, and a general-purpose tool is used to rewrite code for the target application, inserting tests and state to enforce the policy.

Schneider proved that the class of properties that can be enforced by a reference monitor (in-lined or otherwise) is restricted to safety properties[1] and that any enforceable safety property could, at least in principle, be enforced by an execution monitor [46].

In subsequent work, Morrisett, Schneider and their graduate student Hamlen revisited the framework to consider issues of computability and the more general class of *program rewriters* as policy enforcement mechanisms [23]. In this revised model, a program rewriter is allowed to replace faulty or malicious code with any new behavior, as long as that behavior respects the intended policy. Intuitively, IRMs are a special case of rewriters that replace faulty code with "halt". The framework allowed us to compare different enforcement mechanisms, including static analyses,

[1]In the literature on concurrent program verification, *safety properties* correspond to closed sets and *liveness properties* correspond to dense sets. Every property is thus the conjunction of a safety and a liveness property [45]

IRMs, and program rewriters and show formally that rewriters can enforce strictly more policies than the other approaches.

The rewriting approach is particularly attractive from a methodological perspective, because it allows composition and analysis of policies. The conjunction of two policies is enforced by passing the target application through the rewriter twice in succession—once for each policy. By keeping policy separate from program, we can more easily reason about and evolve the security of a system. Furthermore, now that the class of security policies that can be enforced by rewriting is characterized in a mathematically rigorous way, it is ideal for use in connection with other language-based approaches, which exploit formal semantics and analysis.

The rewriting approach also has been shown to be practical. Two generations of prototypes have been built and a third is under construction. The first prototype, SASI (Security Automata SFI Implementation), handled Intel's x86 and Java's JVM architectures [15]; the second, PSLang (Policy Specification Language) and PoET (Policy Enforcement Toolkit), demonstrates the extent to which object-code type annotations are helpful but works only on JVML. The PSLang/PoET prototype gives competitive performance for the implementation of Java's stack inspection security policy [14].

A disadvantage with both SASI and PoET/PSLang is that they depend crucially upon the correctness of the rewriter. Our third prototype, called Mobile and which is being developed for Microsoft's .NET framework, overcomes this limitation. The Mobile rewriter produces modified code along with a set of annotations that allows a third party checker to easily verify that the resulting code respects the policy. Thus, the rewriter is removed from the trusted computing base in favor of a (simpler) trusted type-checker. In this respect, Mobile is a *certified* rewriter that leverages the ideas of proof-carrying code [38].

# 4 Type Systems for Legacy Languages

High-level languages, such as Java, C#, and ML provide compelling security and reliability benefits. In particular, a type-safe language automatically ensures both object-level memory isolation and a form of control-flow isolation, which are necessary to ensure the integrity of any reference monitor. Nevertheless, most of today's security-critical software is written in type-*unsafe* languages. For example, all major operating systems including Windows, Mac OS X, Linux, etc. are written in C. Consequently, attackers have been able to leverage coding flaws including buffer overruns, integer overflows, and format string mismatches—all flaws that would be prevented in a type-safe language—to break into machines.

Given that operating systems consist of tens of millions of lines of code, it is simply too expensive for vendors to throw away the code and start over in a type-safe language. Even if they could afford to do so, there are strong technical reasons that prevent the use of today's type-safe languages in domains such as operating systems, real-time systems, and embedded systems. In large part, this is because today's type-safe languages, unlike C and C++, do not provide the degree of control over data representations, memory management, predictability, and code performance that is needed for these settings.

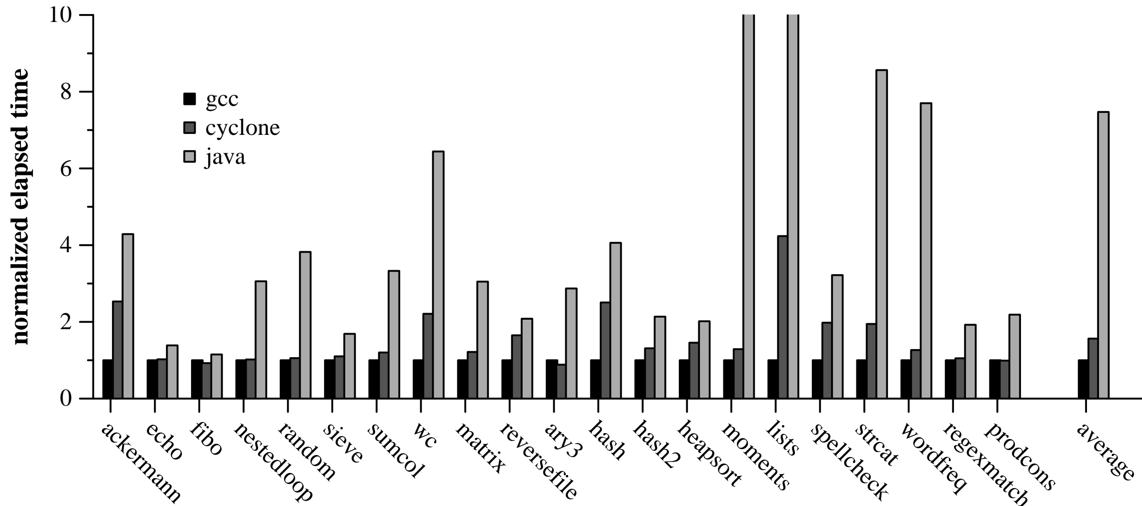


Figure 1: Great Programming Language Shootout Performance for C, Cyclone, and Java

## 4.1 Type-Safe C Code

We have developed a type-safe dialect of C known as Cyclone[2] for use in systems programming. When compared to bug-finding tools such as Lint, SPLint, Prefix and Prefast, the primary advantage of Cyclone is that it makes a strong *guarantee* of type safety that is enforced through a combination of (a) an advanced type system, (b) sophisticated static analyses, (c) language extensions, and (d) run-time checks inserted by the compiler. The type-safety guarantee ensures that a wide class of attacks, including buffer overruns, format string attacks, and integer overflow attacks cannot be used to subvert the integrity of a service. Just as importantly, the static type system ensures that common coding errors (e.g., accessing an uninitialized variable) are caught early.

Of course the safety provided by Cyclone or any other type-safe language has a price. Figure 1 shows the performance of Cyclone and Java code normalized to the performance of C code for most of the micro-benchmarks in the *Great Programming Language Shootout.*[3]

The average over all of the benchmarks (plotted on the far right) shows that Cyclone is about $60\%$ slower than GCC, whereas Sun's Java VM is about $6.5$ times slower. For larger, more realistic benchmarks we see even less overhead for Cyclone. For example, our Cyclone port of the Boa Web server[4], adds only about 3% overhead [25]. And of course, we found and fixed a number of errors in the various programs we have ported, including buffer overruns, that could lead to successful penetrations.

In addition to time, programmers worry about other resources such as space. Most type-safe languages, including Java, take resource control away from the programmer. For instance, memory

[2]The Cyclone compiler, tools, and documentation are freely available at `http://www.eecs.harvard.edu/~greg/cyclone/`.

[3]See `http://shootout.alioth.debian.org/`. The benchmarks were run on a dual 2.8GHz/2GB Red Hat Enterprise workstation. We used Cyclone version 0.8.2 with the `-O3` flag, Sun's Java client SDK build version 1.4.2_05-b04, and GCC version 3.2.3 with the `-O3` flag. Each reported number in Figure 1 is the median of 11 trials.

[4]`http://www.boa.org`

management in Java is handled automatically by a garbage collector. In contrast, Cyclone provides the low-level control over data layout and memory management needed to build kernels and embedded systems. In particular, it provides a sophisticated region-based type system [22] coupled with an ownership model [25] that yields type-safe, real-time, programmer-controlled memory management. In addition, because the language has no hidden type-tags and uses the same data representations and calling conventions as C, it is easy to interface Cyclone code with legacy libraries.

We have found that with respect to space, there are again overheads when using Cyclone as compared to C, but these overheads are much less than for Java. For instance, the C version of the *heapsort* benchmark had a maximum resident working set size of 472 pages, the Cyclone version used 504 pages, and the Java version used 2,471. In general, by supporting safe manual memory management, Cyclone is able to significantly reduce space overheads present in garbage-collected languages [25].

There is another cost to achieving safety, namely, the cost of porting a program from C to a safe language. Porting a program to C# or Java involves a complete rewrite for anything but the simplest programs. In contrast, most of the Shootout benchmarks, and indeed larger programs such as the Boa web server, can be ported to Cyclone by touching 5 to 15 percent of the lines of code. To achieve the best performance, programmers may have to provide additional information in the form of *extended type qualifiers* or *assertions* that are statically checked by the compiler. Of course, the number of lines of code that changed tells us little about how hard it is to make those changes. In all honesty, this can still be a time-consuming and frustrating task. Nevertheless, it is considerably easier than rewriting the program from scratch in a new language.

CCured [39] is another dialect of C that provides a strong type safety guarantee. Like Cyclone, CCured uses a combination of static analysis and run-time checks to ensure object-level type safety. It is generally easier to port an application from C to CCured because the analysis is almost fully automatic. However, CCured implicitly adds meta-data to objects to support run-time checks and relies upon a garbage collector to manage memory. Finally, CCured cannot easily support a multi-threaded environment. These shortcomings make it difficult to use CCured in critical systems such as kernels, embedded, or real-time systems.

Other tools, such as Evans's SPLint [16], Microsoft's Prefix [6] and SLAM [3], and Engler's Metal [12] provide support for sophisticated static analysis or software model checking. These tools can detect bugs at compile time that Cyclone cannot, but most of them, including Prefix, SPLint and Metal are *unsound* because they make optimistic assumptions. For instance, the analyses used in Metal ignore the potential for aliasing or interactions among multiple threads, which is of course common in systems programs. In turn, this may cause the tool to miss a potential bug because the semantics are not accurately modeled. In short, all of these tools are extremely good for finding bugs, but not ensuring their absence. In contrast, Cyclone either reports a static error or inserts a dynamic check for each potential error and can thus provide guarantees.

## 4.2 Type-Safe Machine Code

Java is by far the most widely appreciated example of a language in which type safety is used to provide security. The functionality of a Java-enabled system, such as a Web browser, can be

dynamically extended by downloading Java code (applets) to be executed within the context of the system. In fact, a browser does not accept Java source code directly, but rather, JVML bytecodes (generated by a compiler) that are better suited for direct execution. Therefore, the security of the browser does not depend on the type safety of Java source, but rather the type safety of the executable JVML code.

Before executing potentially malicious JVML code, the browser invokes a verifier to check the type-consistency of the bytecode with respect to an interface that mediates access to the browser internals. The process of type-checking is meant to ensure that the applet code is isolated from the surrounding context of the browser, and that access to browser resources is mediated through the provided interface. Different security policies can be realized by changing this interface. For example, applets from a trusted source (e.g., the local machine) may be given access to the file system, whereas applets from an untrusted source (e.g., another host) may not.

Though more flexible than traditional OS-based mechanisms, the JVML type system is not without shortcomings. First, the type system is relatively weak by modern standards. For instance, it provides no notion of parametric polymorphism. Second, the JVML is a relatively high level CISC machine language tailored for Java. As such, it is ill-suited for compiling other type-safe languages including Cyclone. For example, because JVML provides no support for tail calls, it makes a poor target language when compiling functional languages. Finally, the official specification of the JVML type system is an informal English description and provides no model for ensuring soundness, though recent work has provided formal specifications of important fragments of the language [48, 10, 43, 20, 17, 18, 40]. Even if such formal models can be constructed, it would be a daunting task to prove the correctness of a production JVML verifier, JIT compiler, and runtime. Hence, the JVML model requires trusting a rather large set of components that in practice have proven unreliable [28].

To address some of the shortcomings of the JVML, we have studied and developed advanced type systems for concrete machine languages. A by-product of this work is called Typed Assembly Language (TAL) [31, 32]. Unlike the JVML, TAL is based on a low-level RISC-like machine model and consequently is better suited for compiling a variety of source languages, not just Java. Furthermore, the type system of TAL is powerful enough that many more checks can be done statically and thus compilers can produce more efficient code. For example, a general class of array bounds checks can be statically verified by the current TAL type checker, whereas the JVML type system requires run-time tests.

The current TAL implementation has been tailored for the widely deployed Intel x86 line of processors. On those machines, TAL code can be executed without translation, so the TCB does not include an interpreter or compiler. Like the JVML, key fragments of the TAL type system and semantics have a formal model, and a soundness result has been proven.

# 5 Type Systems for End-to-End Security

Type safety is essential to secure programming. However, code may contain vulnerabilities or malicious code that can lead to security violations without compromising type safety. Malicious programs may simply violate confidentiality (secrecy) by leaking sensitive information. And pro-

grams often contain vulnerabilities that enable attackers to violate confidentiality or integrity. This has been an ongoing problem for various web services. For example, the Hotmail email service has at times had vulnerabilities that permitted users to improperly read each others' mail.

Systems are secure only if they protect the confidentiality and integrity of the data they manipulate. Ideally we would like to be able to state high-level security requirements and have them automatically checked for programs. This would be useful both for users downloading possible malicious programs, and for program designers who want assurance that they have met the security requirements.

We have been exploring policies based on *information flow* (e.g., [19, 11]), which are attractive because they govern *end-to-end* use of information within a system. Information flow policies are therefore more expressive than ordinary (discretionary) access control, which regulates which principals (users, machines, programs, or other entities) can read or modify the data at particular points during execution, but do not track how information propagates.

We have been most interested in enforcing two fundamental information security properties: confidentiality, which requires that information not be released improperly, and integrity, which requires that information be computed properly from trustworthy information sources. To understand where information propagates, it is useful to have access to a program-level representation of the computation using the information. Previous run-time schemes for tracking information flow, such as mandatory access control, lacked precision and imposed substantial time and space overhead.

## 5.1 Mostly-Static Information Flow Control

A compile-time analysis of programs can check that information flows within programs in accordance with confidentiality and integrity requirements. In fact, this static analysis can be described as a type system in which the type of data in the program carries not only information about the structure of the data (such as `int`), but also carries security restrictions on the use of the data. Such a language is said to have *security types*. The program does not have to be trusted to enforce the security policy; only the type checker must be trusted.

Security types are provided by our programming language Jif [34] (for Java Information Flow), which extends the Java language with support for specification and analysis of information flow. The Jif compiler checks information flow, then translates the Jif program to an equivalent Java program with extra embedded annotations that carry security information. Thus, run-time space and time overhead are small. Jif has been publicly available now for a few years and has been used by several other research projects. While a number of other related languages have been designed for theoretical studies (e.g., [50, 24, 54, 4, 42]), Jif remains the most complete and powerful implementation of static information flow analysis, and it has also influenced other language designs.[5]

[5]More information on this approach can be found in our frequently-cited survey of work on language-based information flow security [44].

## 5.2 Security Labels

Jif programs contain labels based on the *decentralized label model* (DLM) [35], in which principals can express ownership of information-flow policies. This model works well for systems incorporating mutual distrust, because labels specify on whose behalf the security policy operates. In particular, label ownership is used to control the use of *selective declassification* [41], a feature needed for realistic applications of information-flow control.

In this model, a *principal* is an entity (e.g., user, process) that can have a security concern. These concerns are expressed as labels, which state confidentiality or integrity policies that apply to the labeled data. Principals can be named in these policies as owners of policies and as readers of data.

For example, a security label specifying confidentiality is written as $\{\mathtt{o}:\mathtt{r}_1, \mathtt{r}_2, \ldots, \mathtt{r}_n\}$, meaning that the labeled data is owned by principal o, and that o permits the data to be read by principals $\mathtt{r}_1$ through $\mathtt{r}_n$ (and, implicitly, o). A label is a security policy controlling the uses of the data it labels; only the owner has the right to weaken this policy.

Labels on data create restrictions on the use of that data. The use of high-confidentiality data is restricted to prevent information leaks, and the use of low-integrity data is restricted to prevent information corruption. The label on information may be securely changed from label $L_1$ to label $L_2$ if $L_2$ specifies at least as much confidentiality as $L_1$, and at most as much integrity as $L_1$. This label relationship is written as $L_1 \sqsubseteq L_2$.

For example, if a Jif program contains variables $x_1$ and $x_2$ with labels $L_1$ and $L_2$ respectively, then an assignment $x_2 = x_1$ is permitted only if $L_1 \sqsubseteq L_2$. Otherwise, the assignment transfers the information in $x_1$ to a location with a weaker security label. For labels in the DLM, this relationship can be checked at compile time, so the labels of $x_1$ and $x_2$ are not represented at run time.

Recently, we have shown how to extend the DLM to include policies for information availability [59]. Because information flow analysis is essentially a dependency analysis, a static information flow analysis can determine how system availability depends on availability of inputs.

## 5.3 Beyond Zero Information Flow

The usual formalization of information security is in terms of *noninterference*, a formal statement that no information flow occurs from one security level to another. Noninterference is mathematically elegant, but real programs need to release some information as part of their proper functioning. For example, a password program leaks a little information about passwords because a guesser learns at least that the password is not the guessed password. To characterize the security of real systems, more notions of security are needed that are more expressive and general than noninterference.

Recent results show substantial progress toward this goal. We identified a property called *robustness* that generalizes noninterference by ensuring that information release, while permitted, cannot be affected by untrusted attackers [53]. Further, we proved that robustness can be enforced by a static program analysis that permits information release only at high-integrity program points [36]. This analysis is built into recent versions of Jif.

Another way to generalize noninterference is to control the quantity of information that is

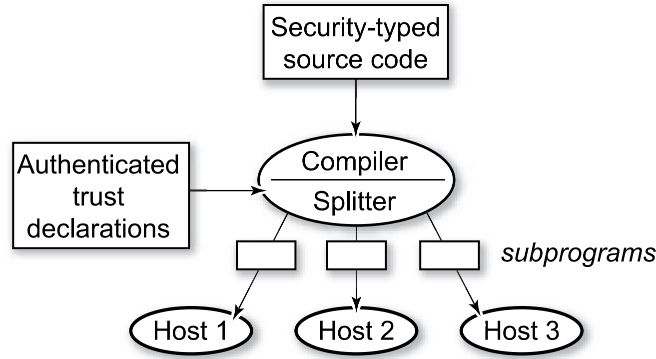


Figure 2: Program partitioning

released by a system. A standard approach has been to model the quantity of information in terms of the reduction in the uncertainty of the attacker [11, 29, 21]. We have shown that *accuracy* can be used a quantitative information metric, avoiding troubling anomalies that arise in the uncertainty-based approach [9].

One further promising approach to generalizing noninterference is to add policies for downgrading and erasure [7, 8]. Downgrading policies say when downgrading can be used to weaken noninterference; dually, erasure policies say when information erasure must be used to strengthen noninterference. Erasure policies require that the system "forget" information at a given security level.

## 5.4 Automatic Partitioning of Secure Distributed Systems

Distributed systems make security assurance particularly difficult, as these systems naturally cross administrative and trust boundaries; typically, some of the participants in a distributed computation do not trust other participants or the computing software and hardware they provide. Systems meeting this description include clinical and financial information systems, business-to-business transactions, and joint military information systems. These systems are distributed precisely *because* they serve the interests of mutually distrusting principals. The open question is how programmers should build distributed systems that properly enforce strong security policies for data confidentiality and integrity.

We introduced automatic program partitioning and replication [56, 57, 58] as a way to solve this problem. As depicted in Figure 2, the Jif/split compiler automatically partitions high-level, non-distributed code into distributed subprograms that run securely on a collection of host machines that are trusted to varying degrees by the participating principals. (Such hosts are *heterogeneously trusted*.) A partitioning is secure if the security of a principal can be harmed only by the hosts the principal trusts. Thus, partitioning of the source program is driven by a high-level specification of security policies and trust.

Both code and data are partitioned to ensure that data and computation are not placed on a machine where confidentiality or integrity might be violated. Sometimes there is no single ma-

chine that is sufficiently trusted to protect the integrity of data or computation; in that case, the partitioning process may replicate the data or computation across several hosts. Results are only considered to be high integrity when all the replica hosts agree on its value.

A number of distributed programs have been implemented using automatic program partitioning and replication, including a variety of online auction programs, a banking simulation, and the game of Battleship. Static information flow checking caught a number of bugs in our implementations of these programs. We compared the distributed programs generated automatically by Jif/split with carefully hand-coded versions of the same programs; the results suggested that the run-time performance of Jif/split programs is reasonable; the hand-coded programs are more efficient primarily because they can exploit concurrency. We have studied information flow in the presence of concurrency [55], but Jif does not yet support concurrent programming because concurrency can create covert timing channels.

# 6 Secure Languages for Firmware

Firmware is low-level driver code associated with hardware devices whose purpose is to provide an interface by which the system can operate the device. On a typical computing platform, firmware is composed of many interacting modules. There is usually some central kernel support, as well as device drivers supplied by the manufacturers of the various hardware components. A driver for a particular device may be used to initialize the device, perform diagnostic checks, establish communication with other devices connected to it, allocate system resources, and other similar tasks. Often the drivers reside in ROM on the devices themselves and are loaded at boot time.

Because these device drivers are normally considered part of the trusted computing base, they constitute a significant security risk. They execute in privileged mode and have essentially unrestricted access to other devices and the entire hardware configuration. They could easily circumvent any operating system-based security mechanism. A malicious driver would have virtually limitless potential to cause irreparable damage, introduce channels for clandestine access, install arbitrary software, or modify the operating system.

Compounding the worry is that most drivers are written by third-party device manufacturers and may come from various subcontractors of unknown origin. Many of these devices and their associated firmware are mass-produced overseas, outside the purview of any domestic authority. It would be well within the capability of a determined adversary to exploit this vulnerability on a massive scale.

Attempts to address this security issue generally fall into two categories: *authentication-based* and *language-based*.

## 6.1 Authentication-Based Approaches

In an authentication-based approach, an attempt is made to ensure the integrity of firmware via digital signatures or chain-of-custody and physical protection. This strategy requires that the firmware was originally benign. This belief is typically based on trust in the supplier or in some detailed inspection of the code by a (human) certifying authority. It simply ensures that the code has not been

tampered with after it was approved. This strategy can preserve an existing relationship of trust, but it cannot establish new trust. Examples of this approach are the driver-certification scheme currently used by Microsoft and the AEGIS system [2].

The authentication-based approach is currently the preferred strategy in practice today. However, its use is not without cost. There may be a large, far-flung network of vendors for whom trust must be established. Moreover, there are mechanisms for automatically updating device drivers and firmware with patches via the Internet. Firmware that is updated regularly needs to be reexamined each time.

## 6.2 The Language-Based Approach

In the language-based approach, firmware modules are written in a type-safe language and compiled to an annotated bytecode form. Each time a firmware module is loaded, it is automatically and invisibly verified against a standard security policy by a trusted verifier. The compiled code and the compiler that produced it need not be trusted. This approach is similar to proof-carrying code and related techniques [37, 38, 30, 33, 27].

In this project, we have developed a prototype called *BootSafe* [1, 49]. The system operates in the context of the Open Firmware standard [26], an IEEE standard for boot firmware that was developed in the mid 1990's and is now in widespread use. Both Sun Microsystems and Apple use boot firmware that conforms to the standard. Several commercial implementations of Open Firmware are available.

The Open Firmware standard is based on the Forth programming language. Currently, device drivers are written in Forth and compiled to *fcode*, a low-level, mostly machine-independent bytecode language similar to the Java virtual machine language. Every Open Firmware-compliant boot kernel must include an fcode interpreter or virtual machine.

The BootSafe architecture consists of several major subsystems:

- J2F, an annotating Java bytecode to Forth fcode compiler;
- a stand-alone verifier;
- an API and runtime support library consisting of various Java classes and interfaces.

BootSafe-compliant device drivers are written in Java. They may extend system classes provided by the API that implement standard functionality and provide access to Open Firmware services. They may also be required to implement certain Java interfaces in the API that specify functionality necessary to conform to the Open Firmware standard. Java thus provides an enforcement mechanism that is absent in Forth.

The Java driver is compiled to bytecode using any standard off-the-shelf Java compiler. The BootSafe compiler J2F is then used to compile the (typed) Java bytecode to annotated fcode. The fcode driver can then be shipped with the hardware device. In Open Firmware, the driver is burned into ROM and stored on the device itself.

When the system boots, the boot kernel recursively probes the system bus to determine the hardware configuration and initialize devices. At that time, each device is probed to see if there is

an on-board driver. If so, it is loaded into main memory, linked against the runtime support library, and executed. Just before execution, the verifier checks the driver to ensure compliance with the security policy.

The security policy used in BootSafe is a standard, baked-in policy appropriate for device drivers. Besides basic memory and control-flow safety, the security policy asserts that device drivers may not access other devices, may only access system memory and bus addresses allocated to them through a strictly controlled allocation and deallocation procedure, and may otherwise interact with the system only through a strict interface provided by the API.

Our prototype contains fully operational and verified Open Firmware-compliant boot drivers written in Java for a network card and a 1.4Mb floppy disk drive.

### 6.3 Limitations

Language-based techniques, while a strong countermeasure to malicious firmware, cannot protect against all forms of attack. For example, certain denial-of-service attacks and malicious hardware are difficult or impossible to detect. However, they do raise the bar by making it more difficult for drivers to operate devices maliciously.

## 7 Putting It Together

In-lined reference monitors, advanced type systems, and certifying compilers are promising approaches to system security. Each allows rich instantiations of the Principle of Least Privilege; each involves only a small and verifiable trusted computing base. Moreover, our language-based security approaches seem ideally suited for use in extensible and component-based software—a domain not served well by traditional operating system reference monitors.

We have solved a large number of both theoretical and practical engineering problems for these separate areas of language-based security. Moreover, we have demonstrated that they can work together to achieve more than the sum of the parts. For example, in the Mobile rewriter, by coupling IRM-style rewriting with certifying compilation, we are able to eliminate the need for a trusted rewriter. By leveraging the type-safety of the Microsoft intermediate language, we are able to avoid the dynamic overheads of protecting the integrity of the in-lined reference monitor. At the same time, we were able to augment the ideas of proof-carrying code to provide for a more dynamic and flexible policy language.

The possible ways to combine rewriting, static analysis, and certification are endless, and the tradeoffs are complex. We believe that a key underlying theme is the relocation of function and trust in the TCB. It seems easier to insert run-time checks than to do analysis before execution is started. Yet, static analysis can lead to better performance, because it occurs offline and deals better with non-safety properties such as availability and integrity. It also seems easier and requires a smaller TCB to check proofs than to create them.

# References

[1] Frank Adelstein, Dexter Kozen, and Matt Stillerman. Malicious code detection for open firmware. In *Proc. 18th Computer Security Applications Conf. (ACSAC'02)*, pages 403–412, December 2002.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proc. 1997 Symposium on Security and Privacy*, pages 65–71. IEEE, May 1997.

[3] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Twenty-Ninth ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, OR, January 2002.

[4] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[5] Brain Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan 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, December 1995.

[6] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, June 2000.

[7] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

[8] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proc. 18th IEEE Computer Security Foundations Workshop*, June 2005.

[9] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proc. 18th IEEE Computer Security Foundations Workshop*, June 2005.

[10] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proceedings of the OOPSLA'98 Workshop on the Formal Underpinnings of Java*, Vancouver, B.C., October 1998.

[11] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[12] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[13] D.R. Engler, M.F. 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.

[14] U. Erlingsson and F.B. Schneider. Irm enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000. To appear.

[15] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, Ontario, Canada, September 1999.

[16] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January 2002.

[17] S. Freund and J. Mitchell. A type system for object initialization in the Java bytecode language. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 310–328. ACM Press, 1998.

[18] S. Freund and J. Mitchell. Specification and verification of Java bytecode subroutines and exceptions. Technical report, Computer Science Department, Stanford University, 1999.

[19] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[20] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. on Computer and Communications Security*, San Francisco, California, October 1998.

[21] James W. Gray, III. Towards a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 21–34, 1991.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[23] Kevin Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*. To appear.

[24] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[25] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory management in cyclone. In *Proceedings of the ACM International Symposium on Memory Management (ISMM)*, pages 73–84, October 2004.

[26] IEEE. *IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*, 1994. Standard 1275-1994.

[27] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Computer Science Department, Cornell University, January 1998.

[28] Gary McGraw and Edward Felten. *Hostile Applets, Holes and Antidotes*. John Wiley and Sons, New York, 1996.

[29] Jonathan K. Millen. Covert channel capacity. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1987.

[30] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[31] Greg Morrisett, David Walker, Karl Crary, and Neal 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.

[32] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[33] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[34] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, TX, USA, January 1999.

[35] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[36] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proc. 17th IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[37] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[38] George C. Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Programming Languages*, pages 106–119. ACM SIGPLAN/SIGACT, January 1997.

[39] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[40] R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 70–78, January 1999.

[41] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.

[42] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[43] Zhenyu Qian. A formal specification of Java(tm) virtual machine instructions for objects, methods, and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java(tm)*. Springer Verlag LNCS, 1998.

[44] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[45] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[46] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2(4), March 2000.

[47] Margo Seltzer, Y. Endo, Chris Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[48] Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, January 1998.

[49] Matt Stillerman and Dexter Kozen. Demonstration: Efficient code certification for open firmware. In *Proc. 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*, volume 2, pages 147–148. IEEE, IEEE Computer Society, Los Alamitos, CA, April 2003.

[50] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[51] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 203–216, Asheville, December 1993.

[52] E. Yasuhiro, J. Gwertzman, M. Seltzer, C. Small, Keith A. Smith, and D. Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard Computer Center for Research in Computing Technology, 1994.

[53] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.

[54] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *Proc. 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, 2001.

[55] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

[56] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, pages 1–14, Banff, Canada, October 2001.

[57] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[58] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003.

[59] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *Proc. 18th IEEE Computer Security Foundations Workshop*, June 2005. To appear.