Successful attacks on computing infrastructures often involve failures of type safety. A major contribution of this grant has been the creation of type systems and type-checking algorithms for low-level languages in use today. In addition, "certifying compilation" was developed to eliminate the need to trust correctness of high-level language implementations.

However, ensuring type safety is not sufficient for ruling-out misbehavior in code. A second contribution of this grant was to design and build program-rewriting tools employed for security policy enforcement and also to derive a theoretical characterization for what kinds of policies can be enforced by program rewriting. The theoretical work compares the expressive power of rewriting against traditional security enforcement mechanisms; rewriting is proved to be strictly more powerful. The in-lined reference monitor toolkits handle x86 machine code, the Java virtual machine, and Microsoft's .NET framework.
Objectives

To better support flexibility, evolution, and performance, a new class of system architectures is emerging. Integral to these newer architectures is support for clients to extend service interfaces dynamically. Specifically, clients can send code extensions—perhaps over a network—to services, and these services execute this foreign code on behalf of the client.

Unfortunately, the flexibility provided by extensible architectures is also a source of vulnerability, as misbehaved extensions can cause considerable damage. Extensible systems therefore must have security mechanisms to protect against malicious actions by foreign code.

We need mechanisms that support enforcement of a rich class of security policies for extensible systems. The mechanisms should have modest runtime overhead or else a primary attraction of extensible architectures will be lost. And the tension between flexibility and performance is what makes this security problem a particularly difficult one to tackle.

This project studied how programming language technology could be leveraged to support enforcement of rich classes of security policies. We identified and exploited ideas from programming language design, type and
proof systems, semantics, and implementation that provide a basis for meeting the twin goals of flexibility and performance.

Status of Effort

The research supported by this grant developed two major areas of language-based security: (1) Type systems for verifying the memory safety of systems code, and (2) Flexible enforcement of security policies through program rewriting.

Many of the successful attacks on our computing infrastructure involve a failure of type safety. To date, most type-safety analyzers have depended upon the code being written in a high-level, structured language (e.g., Java). But the vast majority of our computing infrastructure is coded in low-level languages such as C. One of our major contributions was the development of type systems and type-checking algorithms for such low-level languages. In addition, we developed a technique called certifying compilation that eliminates the need to trust that a high-level language’s implementation is correct.

Ensuring type safety is necessary but insufficient to rule out misbehavior in code. We therefore explored a new approach to enforcing desirable behavior based on rewriting executable code. The essence of the approach is that one can express a high-level policy in a declarative language and our rewriting tool in-lines a reference monitor into the binary which enforces that policy. Our research in this area included fundamental theoretical results as well as practical tools. In particular, we developed theoretical models that let us compare the expressive power of rewriting when compared to traditional security enforcement mechanisms and showed that rewriting is strictly more powerful. We also developed in-lined reference monitor toolkits for machine code, the Java virtual machine, and Microsoft’s .NET framework.

Accomplishments/New Findings

Our research concentrated on the following areas of language-based security:

- Type systems for verifying the safety of low-level code.
- Inlined reference monitors for enforcing security policies.
Type Systems for Low-Level Code

Today, our computing and communications infrastructure is built using unsafe, error-prone languages such as C or C++ where buffer overruns, format string errors, and space leaks are not only possible, but frighteningly common. In contrast, type-safe languages, such as Java, Scheme, and ML, ensure that such errors either cannot happen (through static type-checking and automatic memory management) or at least are caught at the point of failure (through dynamic type and bound checks). This fail-stop guarantee is not a total solution, but it does isolate the effects of failures, facilitates testing and determination of the true source of failures, and it enables tools and methodologies for achieving greater levels of assurance.

The obvious question is: “Why don’t we re-code our infrastructure using type-safe languages?” Though such a technical solution looks good on paper, the cost is simply too large. For instance, today’s operating systems consist of tens of millions of lines of code. Throwing away all of that C code and reimplementing it in, say Java, is simply too expensive.

Under the auspices of this grant, we have explored how to adapt type systems to low-level languages, such as C/C++ and even assembly language. The goal has been to (a) provide effective tools that allow current systems to be statically or dynamically checked to ensure type safety, and (b) to eliminate the need to trust those tools through the process of certifying compilation.

Cyclone Compiler

As a part of this research, we developed Cyclone, a type-safe extension to the C programming language. The type system of Cyclone accepts many C functions without change and uses the same data representations and calling conventions as C. The Cyclone type system also rejects many C programs to ensure safety. For instance, it rejects programs that perform (potentially) unsafe casts, that use unions of incompatible types, that (might) fail to initialize a location before using it, that use certain forms of pointer arithmetic, or that attempt to do certain forms of memory management.

All of the analyses used by Cyclone are local (i.e., intra-procedural) so that we can ensure scalability and separate compilation. The analyses have also been carefully constructed to avoid unsoundness in the presence of threads. The price paid is that programmers must sometimes change type
definitions or prototypes of functions, and occasionally they must rewrite code.

We find that programmers must touch about 10% of the code when porting from C to Cyclone. Most of the changes involve choosing pointer representations and only a very few involve region annotations (around 0.6% of the total changes). So, we developed a semi-automatic tool that can be used to automate most of these changes.

The performance overhead of the dynamic checks depends upon the application. For systems applications, such as a simple web server, we see no overhead at all. This is not surprising, as these applications tend to be I/O-bound. For scientific applications, we were seeing a much larger overhead (around 5x for a naive port, and 3x with an experienced programmer), due to array bounds and null pointer checks. To avoid these, we incorporated a sophisticated intra-procedural analysis that eliminates most of those checks. For instance, a simple matrix-multiply now runs as fast as C code, where before, it was taking over 5x as long.

We also introduced new typing mechanisms that support a wide range of safe memory management options. Initially, we had to restrict programmers to using only garbage collection, stack allocation, or limited forms of region allocation, all of which could adversely affect time and space requirements. But we have since added support for dynamic region allocation, unique pointers, and reference-counted objects. These mechanisms let programmers control memory management overheads without sacrificing safety. For instance, we were able to improve the throughput of the MediaNet streaming media server by up to 42% and decrease the memory requirements from 8MB to a few kilobytes using these new features.

Cyclone is actively used by the research community to ensure safety for real systems code. For instance, AT&T researchers are using Cyclone to develop a number of high-confidence systems; researchers at Washington and Utah are using Cyclone to develop extensible protocols; and researchers at the Leiden Institute have used Cyclone to develop secure kernel extensions for Linux. They have found Cyclone attractive because the programming model is close to C but provides the strong safety guarantees need for secure systems.
Typed Assembly Language

Type safe languages such as Cyclone can, in principle, provide strong security guarantees. However, the Cyclone compiler and the associated tools are well over 200,000 lines of code. It is likely that there are bugs in these tools which could be exploited by an attacker. Cyclone is not alone in this regard—the type safety of any language (e.g., Java) depends upon the correctness of the implementation of that language, including the compiler or interpreter, the libraries, and the run-time system. These software systems are large and experience has shown that we cannot depend upon them being 100% correct.

The goal of the Typed Assembly Language (TAL) research was to eliminate the need to trust language implementation tools. In particular, the TAL project developed a type system for Intel x86 machine code and a type-checker which consisted of roughly 20,000 lines of code. With the TAL type-checker, it becomes possible to check that a compiler for a high-level language, such as Cyclone or Java, is producing code that actually is type-safe. Once again, we must trust the TAL type-checker, but it is an order of magnitude smaller than the Cyclone compiler (and two orders of magnitude smaller than Sun’s Java implementation) which no longer needs to be trusted.

The primary challenge in developing TAL was finding a set of type constructors that supported compilation of a wide variety of source programming languages. To keep the type system small but flexible, we adapted a suite of higher-order type constructors which could be combined to build higher-level typing abstractions. For instance, TAL had no built-in notion of procedure call and return. Rather, it had simple type constructors for describing machine states at each program point and these type constructors could be composed to specify typing pre- and post-conditions for procedures. This degree of flexibility was crucial for supporting a wide variety of languages.

Inlined Reference Monitors

Inlined reference monitors (IRM) are a new approach to implementing traditional reference monitors. A desired end-to-end security policy is formulated using a high-level declarative policy language, and then a rewriting tool is used to automatically rewrite untrusted code into code that respects the policy. The rewriting tool works by inserting extra state and dynamic checks into the untrusted code so that the code becomes self-monitoring.
Under the period of this funding, our two PSLang/PoET implementations of Java 2 stack inspection were completed and analyzed. We reproduced Wallach’s “security passing style” implementation of the stack inspection policy and obtained comparable performance, and we devised a new implementation of the policy and obtained superior performance. The new implementation works by carefully allocating work so that frequently executed JVM instructions bear relatively less of the burden associated with enforcement. The implementation exhibits performance that is competitive with the JVM-resident stack inspection implementation included in the commercial Java distribution.

We also implemented a prototype deployment of an IRM for a production operating system. Specifically, a set of kernel modifications was developed to support a prototype IRM rewriter in Microsoft’s Windows. This work suggests the need for mechanism to identify which policy is applied to any given executable and for mechanism to manage multiple executables (each enforcing a different policy). For example, .NET caches dll’s (executables), and the architecture for how that cache is managed needs to work differently when the same dll could have been rewritten in multiple ways (to enforce one or another different policies).

In addition, a prototype MSIL (Microsoft Intermediate language) inlined reference monitor realization was completed. It implements an aspect-oriented programming metaphor for MSIL assembly language (rather than for a high-level language). An aspect-oriented program comprises aspects, each of which consists of a point-cut and some advice. The point-cut is a predicate that specifies where to do rewriting in target code, and the advice specifies how to do the rewriting. Designing a point-cut language that provides complete visibility at a high-level into an assembly language is an interesting challenge. We subsequently extended this prototype so that we could perform arbitrary rewriting on the CIL code by building on a bytecode-rewriting toolkit developed by Microsoft Researchers.

Working with Ph.D. student Kevin Hamlen, we developed a more refined characterization of what policies can be enforced using reference monitors. This new work extends earlier work by Schneider, now taking into account the limits of computability. Specifically, we developed a model based on standard Turing machines, adapted Schneider’s criteria for enforceable security policies, and introduced computability requirements. We also integrated static analysis and program rewriting into the model.

By providing this unifying model, and by basing it on Turing machines,
we were able to compare the relative power of the various enforcement mechanisms, and to relate them to standard computability results. For instance, it was relatively easy to show that the class of policies precisely supported by static analysis could also be supported by both reference monitors and by program rewriting. In addition, we found that introducing a computability requirement on reference monitors was necessary, but not sufficient, for precise characterization of the class of policies actually realizable by reference monitors. And we identified a new property, which we call “benevolence” that provides a more accurate upper bound on the power of reference monitors.

Our most surprising and important results involve program rewriting. We can show that the class of policies originally characterized by Schneider does not include all policies enforceable through rewriting (and vice versa). Indeed, we were able to show that the class of policies enforceable through rewriting does not correspond to any class of the Kleene hierarchy. This is a surprising and important result, as it shows that rewriting truly is a powerful security enforcement technique.

**Personnel Supported**

**Faculty:** Greg Morrisett and Fred B. Schneider.

**Postdoctoral Researchers:** Amal Ahmed, Mike Hicks, Yaron Minsky, Mike Marsh.

**Graduate Students:** James Cheney, Ulfar Erlingsson, Neal Glew, Daniel Grossman, Kevin Hamlen, Yaron Minsky, Frederick Smith, David Walker, Stephanie Weirich, Steve Zdancewic, and Lidong Zhou.

**Publications**


**Interactions/Transitions**

**Presentations at Meetings, Conferences, Seminars, etc**

**Invited Lectures: F.B. Schneider**


42. The Case for Language-Based Security. Department of Computer Science, University of Tromso, Tromso, Norway, March 2002.


Invited Lectures: G. Morrisett


Consultative and Advisory Functions

- Schneider chaired a study for DARPA IPTO Program Manager Jay Lala on promising research directions for Self-Healing Networked Information Systems.

- As a consultant to DARPA/IPTO, Schneider chaired the independent evaluation team for the OASIS Dem/Val prototype project. This project funded two consortia to design a battlespace information system intended to tolerate a class A Red Team attack for 12 hours.

- Microsoft researchers collaborated with Morrisett on the design and implementation of a low-level, type-safe language for building device drivers.

- Greg Morrisett spent nine months visiting Microsoft’s Cambridge Research Laboratory, where he worked with researchers on programming language and security technology. In particular, Morrisett worked on the development of Microsoft’s tools for automatically finding security flaws in production code, based on his experience with Cyclone. He also worked with student Kevin Hamlen and Microsoft researchers on the implementation of the .NET rewriting tool for inline reference monitors.

Transitions

- Researchers at Carnegie-Mellon University, Princeton University, University of California (Riverside), University of Newcastle-Upon-Tyne, and Intel Research are all now building on PoET/PSLang IRM tools developed by Schneider and collaborators.
• AT&T research collaborated with us to develop the Cyclone language, compiler, and tools. In addition, researchers at the University of Maryland, the University of Utah, Princeton, and the University of Pennsylvania, and Cornell are all using Cyclone to develop research prototypes.

• Researchers at Leiden Institute of Advanced Computer Science in the Netherlands have developed an extension of the Linux operating system, whereby untrusted modules, written in Cyclone, can be dynamically loaded and executed in the context of the kernel.

New discoveries, inventions, or patent disclosures
None.

Honors and Awards

F.B. Schneider:
• Fellow, American Association for Advancement of Science (1992).
• Fellow, Association for Computing Machinery (1994).
• Professor-at-Large, University of Tromso, Tromso, Norway (1996–2004).
• Daniel M. Lazar Excellence in Teaching Award (2000).
• Doctor of Science (honoris causa), University of NewCastle-upon-Tyne (2003).

G. Morrisett:
• Sloan Fellow (1998).
• NSF Faculty Early Career Development (1999).
• Presidential Early Career Award for Scientists and Engineers (2000).
• Ralph Watts Excellence in Teaching Award, Cornell University (2001).
• Allen B. Cutting Chair of Computer Science, Harvard University (2004).