The goal of this work is to explore techniques for making today's software, which is largely written in type-unsafe, low-level languages such as C, as reliable and trustworthy as code written in type-safe, high-level languages such as Java or ML. Type-safe languages automatically block or prevent common vulnerabilities such as buffer overruns, format string attacks, and overflow attacks which are all too common in today's critical software infrastructure. To this end, we have implemented a prototype compiler called Cyclone, which provides the benefits of type safety through a combination of static analysis, programmer annotations, and run-time checks. Particular emphasis has been placed on scalable, static analyses to ensure that programmers can retain good performance and high reliability.
Objectives

High-level programming languages, such as Java and C#, can increase the reliability of systems and drastically cut development costs. One reason is that the type checkers of these languages automatically identify common programming errors at compile time. Another reason is that the type-safety guarantee of such languages ensures that all primitive operations have a well-defined semantics, thus bounding the space of possible behaviors of a program with coding errors in it.

However, the operating systems, device drivers, network software, servers, and databases that power our computational infrastructure are all written in low-level, unsafe languages such as C or C++. Even the run-time systems, garbage collectors, JIT-compilers, and much of the libraries for Java and C# are coded in C. Consequently, these systems suffer from a variety of preventable flaws that can cost time, security, money, and even lives. As one example, over 50% of the successful attacks reported by the Computer Emergency Response Team (CERT) involve buffer overruns—an attack that
leverages the lack of type-safe arrays in C and C++. The Code Red and Blaster worms are examples of malicious code that used buffer overruns to cause widespread damage.

One reason for this state of affairs is that today’s software systems started out in unsafe, low-level languages and have been evolved over many years. At this point, it is too expensive to throw out the existing code and replace it with code written in a high-level language. For example, Windows Vista contains over 60 million lines of code, much of which was inherited from Windows XP, which was in turn evolved from Windows NT. Microsoft simply cannot afford to pay the development costs needed to re-code an operating system that has been evolved over fifteen years.

Even if Microsoft could afford to re-code Vista in a type-safe language, there are substantial technical issues that prevent the adoption of today’s high-level language technology for building production systems. In particular, the very mechanisms used to enhance safety prevent programmers from:

1. achieving high performance and low resource consumption,
2. interoperating with legacy systems,
3. directly accessing and controlling memory and devices, and
4. reasoning about the space or time requirements of programs.

Thus, even if we could start from scratch, we would have no effective, safe environment for re-coding our infrastructure.

Therefore, the goal of this project was to explore the design, implementation, and foundations of safe, low-level programming languages. The primary aim was to gain a deeper understanding of how programmers can gain control over the performance and interoperability of systems code without sacrificing type safety. In addition, the project aimed to introduce new language-level invariants and enforcement tools that go well beyond simple types and that can be used to build systems that are truly robust.

To evaluate the effectiveness of proposed mechanisms, the project built a prototype, next-generation systems language called Cyclone. Cyclone is based upon ANSI C so that the language can be used and evaluated in real systems contexts. In particular, the project ported and evaluated a number of medium sized programs from C to Cyclone. Basing the language upon C also ensured that systems programmers would be comfortable using the
language, and that interoperability with legacy C (and C++) code would be relatively simple.

Unlike C, Cyclone comes equipped with an advanced type and assertion system that (a) allows programmers to state and enforce crucial program invariants, (b) lets tools automatically check for safety and security properties, (c) gives the compiler sufficient information to produce high-performance code. All of the sources and documentation for Cyclone have been made freely available to the public (with an open-source license) and can be found on the Web at http://cyclone.thelanguage.org.

Status of Effort

Normal C programs can crash in any number of ways due to at least the following:

- unsafe casts
- unsafe pointer arithmetic
- dereferencing a pointer to deallocates storage
- failing to deallocate storage
- failure to allocate large enough buffers
- failure to test for NULL pointers
- unsafe unions

Cyclone prevents all of these problems (and more) through a combination of static and dynamic checks [7]. For instance, all pointer arithmetic in Cyclone is checked to ensure that pointer dereferences lie within the boundaries of a given object.

The Cyclone type checker rules out many simple errors, such as unsafe casts, at compile-time. The key challenge was ensuring that the the type-checker does not reject common idioms that are safe, but use potentially unsafe primitives. To this end, Cyclone supports various forms of polymorphism including parametric polymorphism, existential polymorphism, and physical
subtype polymorphism. Many subtelties arise in the interaction of polymorphism with the imperative features in C (e.g., the address-of-operator) [1], and thus great care had to be taken to ensure soundness.

The Cyclone type checker also rules out many simple memory management errors, such as dereferencing a pointer to a stack-allocated object which has been deallocated. To do so, Cyclone employs a sophisticated region-based, type-and-effect system [6] as well as a form of linear (or tracked) pointers [19,24]. In essence, the type checker conservatively tracks the set of locations that have not been deallocated at each program point. Programmers must provide annotations on procedure boundaries and data structures to give the type-checker enough information to be effective. However, in practice, the annotations are quite small and can be inferred with an external tool [19].

Some problems, such as an array index out of bounds, are prevented by run-time checks inserted by the compiler. To support these checks, the Cyclone compiler must also insert extra meta data on key objects so that the bounds can be determined dynamically. Constructing and maintaining the meta data is a serious source of ineffeciency and thus the Cyclone compiler performs a number of analyses to minimize the overheads.

Finally, the Cyclone type checker provides limited support for extended static checking, wherein programmers can specify pre- and post-conditions on procedures, and static assertions on loops. The compiler uses these pre- and post-conditions to refine its analysis so that most checks can be safely eliminated. For example, when bootstrapping the compiler (which is itself written in Cyclone), over 92% of the NULL-pointer and array-bounds checks are automatically eliminated.

The project ported both small benchmarks as well as complete applications to Cyclone to determine the overheads introduced by the compiler [20]. On small benchmarks, the Cyclone code is on average about 60% slower than unsafe C code. In contrast, Java is about 650% (over an order of magnitude) slower than C code. For more realistic benchmarks, the overheads of Cyclone do not seem to be a problem. For example, when ported from C to Cyclone, the Boa web server came within 2% of the C performance on throughput tests.
Accomplishments

The key technical accomplishments of this project are as follows:

- Adapted parametric and subtype polymorphism to low-level C code; constructed models and proofs of soundness for the resulting type system. In so doing, the project uncovered a number of subtle, previously unrecognized issues when parametric polymorphism is combined with pointer arithmetic.

- Designed, modeled, proved correct, and implemented a region-based, type-safe memory management system. This part of the type system ensures that code respects the lifetimes of data that are allocated on the stack or in lexically-scoped regions. In practice, this part of the type system was crucial for avoiding the overheads of heap-allocation when porting code from C to Cyclone and accounts for a relatively large part of the performance win of Cyclone over existing safe languages.

- Designed, modeled, proved correct, and implemented extensions to the region-based type system to support linear (a.k.a. tracked) pointers. The linear pointers in Cyclone allow programmers to have a much finer degree of control over the lifetimes of heap-allocated data. They can be used to safely implement a wide range of memory management strategies, including standard malloc/free, arenas, and reference counting. With the addition of these facilities, Cyclone became the first type-safe language that could be used to build a garbage collector within the language, and without itself needing a meta-level garbage collector [14].

- Designed, modeled, proved correct, and implemented a static extended checker for Cyclone that ensures certain run-time checks, which are needed to ensure type safety, can be safely eliminated at compile time. The extended static checker runs as fast as a conventional type-checker, and yet statically verifies a range of properties that are outside the scope of traditional type-checkers including array bounds checks and NULL-pointer checks.

- The project produced four Ph.D.'s (J.Cheney, D.Grossman, M.Flue, and S.Weirich.) Of these, three now have top faculty positions at leading universities (Grossman is at Univ. of Washington, Fluet at Toyota...
Technical Institute, and Weirich at Univ. of Pennsylvania) and one is a post doctoral candidate (Cheney at Univ. of Edinburgh.) In addition, the project supported two post doctoral candidates who now have faculty positions (M. Hicks at Univ. of Maryland, and A. Ahmed at Toyota Technical Institute.) Finally, the project supported two undergraduate students who are now in top Computer Science graduate programs (Francis Spaulding at Princeton, and Daniel Lee at Carnegie Mellon.)

**Personnel Supported**

Faculty: Greg Morrisett

**Post Doctoral Students:** Michael Hicks, Amal Ahmed

Graduate Students: James Cheney, Daniel Grossman, Matthew Fluet, Yanling Wang, Stephanie Weirich.

**Undergraduate Students:** Frances Spaulding, Daniel Lee.

**Publications:**

Copies of the publications can be found at the PI's web site: http://www.eecs.harvard.edu/~greg.


25. Kevin Hamlen, Greg Morrisett, and Fred B. Schneider. Certified In-Lined Reference Monitoring for .NET. ACM SIGPLAN Workshop on
Programming Languages and Security, (PLAS), (Ottawa, Canada, June 2006).


**Interactions/Transitions**

- Trevor Jim of AT&T research worked with us to develop the Cyclone language, compiler, and tools. In addition, researchers at the University of Maryland, the University of Utah, Princeton, the University of Washington, and the University of Pennsylvania, and Cornell are all using Cyclone to develop research prototypes. Papers describing systems built with Cyclone have started to appear at major conferences, including IEEE OpenARCH, International Working Conference on Active Networks, and SOSP.

- Morrisett serves on the Microsoft Trustworthy Computing Academic Advisory Board. This group, which meets twice annually in Seattle, advises Microsoft regarding a range of security and privacy issues.

- Morrisett serves on the DARPA ISAT Advisory Board.

- Morrisett serves on the Fortify Technical Advisory Board. Fortify has adapted some of the ideas behind Cyclone’s analysis in their product, Fortify Source Code Analysis.

- Morrisett serves on the editorial boards for: *Journal of Functional Programming* (chief editor), *Information Processing Letters, ACM Transactions on Programming Languages and Systems*. He also serves on
the Advisory Committee for the Semantics, Applications, and Implementations of Program Generation (SAIG) conference, and the ACM Conference on Types in Language Design and Implementation.

• 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.

• Morrisett has also worked with researchers Martin Abadi and Ulfar Erlingsson at Microsoft’s Silicon Valley Research Lab.

Invited Lectures: G. Morrisett


27. Making C Type-Safe. Purdue University, West Lafayette, ID. October 2005.


29. Static Extended Checking for Cyclone. ITU University, Copenhagen, Denmark, May 2006.


Honors and Awards

G. Morrisett:

- NSF Faculty Early Career Development (1999).
- Presidential Early Career Award for Scientists and Engineers (2000).
• Ralph Watts Excellence in Teaching Award, Cornell University (2001).