Bratin Saha

Summary of my Dissertation Research

The security of information systems is becoming critical, especially with the increasing dependence of businesses and individuals on highly networked computing systems. Existing systems enforce security either by authenticating programs, or by monitoring program execution. Authentication is useful in establishing ownership and fixing responsibility, and can be performed statically. It does not, however, give any guarantee about the runtime behavior of programs. On the other hand, program monitoring does guarantee that malicious code will not be executed; but it incurs a significant runtime penalty since it requires extensive dynamic checks, or uses the protection mechanism of the underlying hardware and operating system. Furthermore, a malicious program may have to be aborted after it has changed state or acquired resources. Moreover, hardware and operating system based protection may not be feasible in a resource-constrained environment, such as in portable devices.

Certifying compilation is a new method for implementing secure systems that combines the key features of the previous approaches: it statically checks and guarantees the safety of a program. A certifying compiler generates not only the object code, but also a proof that the code satisfies a security policy. To ensure compliance with the policy, a code consumer only needs to check that the proof is consistent with the code. The checking can be done off-line, and therefore, does not incur a runtime penalty. Moreover, a code consumer does not need to trust the source of the program, or the compiler generating the code. It only needs to trust the checker which is much smaller and simpler (and hence easier to verify) than the compiler.

In my dissertation research, I have developed a new framework for generating low-level certified code. The framework improves upon the state of the art in certifying systems in the following two ways. First, it includes constructs to certify runtime services. Second, it integrates an entire proof system into a compiler intermediate language.

The reliability of a computing platform depends critically on the safety of the runtime services provided by the host system. These services consist of functions like the garbage collector and the linker which are complex pieces of code and often introduce subtle bugs. Verification of these services will increase the security of a system considerably, and hence is an important goal of certifying compilation. Unfortunately, it is a very hard problem and therefore, existing certifying systems rely on trusted runtime services for their safety. These services analyze types (to various degrees) at runtime. Moreover, they may analyze the type of any runtime value. Researchers in type-directed compilation had worked on runtime type analysis. The proposed solutions, however, had significant limitations and cannot be used to certify functions like a garbage collector. Analyzing the type of arbitrary runtime values, such as polymorphic code blocks and function closures, still remained an open problem. Using parametricity, I have designed a new type system that removes the limitations of previous approaches and solves the problem in its full generality. The result is described in the paper “Fully Reflexive Intensional Type Analysis” which appeared in the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000).

To demonstrate the expressive power of the framework, I have shown that it can be used to write a provably type-safe stop-and-copy garbage collector. Proving the type-safety of a garbage collector was long considered to be one of the most challenging problems in certifying compilation. Although I considered only a prototypical garbage collector, the solution handles forwarding pointers satisfactorily, works in a system with separate compilation, and can also be extended to generational collectors. The solution is shown in the paper “Principled Scavenging” which appeared in the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2001).

Current compilers that generate certified code have focused only on traditional type safety. The intermediate language in my framework goes a step further by supporting the explicit representation of proofs, propositions, and inductive reasoning. This is significant since it allows many program invariants, that are left implicit now, to be expressed in the language and checked mechanically by a verifier. Therefore, we can statically enforce more sophisticated program properties. For example, one can ensure statically that unchecked array accesses are safe. The intermediate language is described in the paper "A Type System for Certified Binaries" which will appear in the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2002).

I have implemented a prototype version of the framework in the FLINT-SML/NJ compiler. To make the implementation efficient, I have used hash-consing and memoization techniques extensively. Although the new compiler uses a vastly more expressive type system, the size of types generated and the compilation time increases only by a factor of 2 over the current implementation. I have also developed a new optimization that forces all type manipulation to occur at link time. This ensures that programs do not incur a runtime penalty due to type passing. The implementation results will appear in my dissertation.