Research Statement for Steve Zdancewic

My research draws on the areas of logic, programming-languages theory, compiler technology and security to provide ways to reason about software. My goal is to develop tools and techniques for building safe, reliable, and secure systems. In this statement, I first overview my past and current research and then discuss some intended future directions.

Previous and Ongoing Research

Our society is increasingly dependent on computers and the Internet, whether for e-mail services, web-based shopping and financial planning, business-to-business transactions, or military information systems. This dependence makes software reliability and information security a critical concern. If we are to provide security for on-line data, the sheer complexity of today's software necessitates tools to help automate the process. One of the goals of my research is to build these tools and explore the underlying principles on which they should be based.

Establishing Program Properties via Type Abstraction

Software designers use abstractions to hide implementation details and factor large programs into manageable pieces. Programming languages provide built-in abstractions such as functions, objects, structures, and pointers, from which programmers build their own abstractions such as file handles, linked lists, and button boxes. It is crucial to guarantee that these abstractions are not violated, because such violations may crash the system or damage its integrity. For example, file handles and other system resources should not be forged or otherwise misused by application.

Advanced programming languages such as ML and Haskell (and soon, Java) include *parametric polymorphism*, which allows the user to define programs that operate generically over data, independent of its underlying representation. Parametric polymorphism gives a strong mechanism for proving properties about the abstractions used in a program: For instance, it could be used to show that file handles cannot be forged by application-level software and instead must be provided by the operating system. Unfortunately, with this power comes a price: we must somehow establish that the programming language satisfies the parametricity condition. In complex, realistic languages, this can be quite a difficult task—proving parametricity with some combinations of language features is still an open problem.

In joint work with Greg Morrisett and Dan Grossman [1, 2], I developed an alternative proof technique that establishes weaker properties than full parametricity, yet is still powerful enough to prove interesting properties about abstractions, including this file-handle example. Our conference paper, which describes the technique and some of its applications, won a Best Paper Award at the Principles, Logics and Implementations of High-Level Languages 1999 colloquium.

Information Security

My recent research focuses on just one aspect of information security: protecting the confidentiality and integrity of electronic data. The approach is to use *security-typed languages* in which type systems include mechanisms for specifying policies on how programs manipulate confidential data. Just as conventional compilers for languages like Java verify properties of the program by analyzing user-supplied type annotations, the compiler for a security-typed language determines whether a program obeys its programmer-specified security policy. Programs with security violations are rejected at compile-time, before they have a chance to cause trouble.

I have collaborated with my advisor, Andrew Myers, and others on the design and implementation of a security-typed variant of Java called Jif. Jif supports (almost) all of Java's features and, in addition, provides a rich language for describing confidentiality and integrity policies, resulting in a practical tool for developing secure software. It is available on the project website at `http://www.cs.cornell.edu/jif`. Jif has also served as a test bed for new ideas and as a source of motivating problems addressed by my research:

1. *Compiling secure programs*: One difficulty with compiling security-typed languages is the possibility that the compilation process itself may introduce security holes, perhaps due to bugs in the compiler. A promising way to address this problem is via *proof-carrying code*, in which a program is accompanied by a proof that it satisfies a desired property. The idea is that the compiler, in addition to producing low-level code (assembly language or bytecode, for instance), produces a proof that this low-level code satisfies the high-level security policy. To do so, the compiler essentially must be able to transform a high-level proof of the property into a low-level one.

   In the case of information flow, a naive approach to compilation requires the introduction of conservative approximations that cause the low-level proof to fail; even though the program should be accepted, it is rejected as insecure. To rectify this problem, I developed a more accurate way of proving information-flow security in low-level languages [3, 4]. This improved precision, which can easily be implemented by the checking algorithms in a compiler, provides a way of verifying information-flow properties at the assembly-language level. Perhaps more importantly, the technique, which is based on concepts from linear logic, appears to apply also to the more general problem of information-flow security in concurrent systems.

2. *Characterizing security policies*: Realistic information-flow policies often require declassification, for instance to allow the transmission of encrypted, confidential data over the Internet. Unfortunately, declassification operations can easily result in the accidental release of confidential information, which makes it difficult to reason about the properties of security policies involving declassification. As a first step towards understanding these security policies, I have developed the concept of *robust declassification* that distinguishes between intentional and unintentional release of secret data [5]. This model reveals a connection between the integrity and the confidentiality of the data being manipulated by programs that use declassification.

3. *Heterogeneous trust in distributed systems*: Previous research on security-typed languages has addressed information-flow security in programs running on a single trusted host, or distributed among equally trusted hosts. This assumption is unrealistic, particularly in scenarios for which information-flow security policies are most desirable—when multiple principals need to cooperate but do not entirely trust one another.

   In an award-winning paper presented at SOSP 2001 [6], my colleagues and I have shown how the compiler can use the security policy itself to automatically partition a program among a collection of heterogeneously trusted hosts. The resulting distributed system implements the original program, but, in addition, satisfies the security policies of all participants without requiring a universally trusted host.

FUTURE RESEARCH

In the context of information security, there are a number of specific, immediate questions that I am interested in pursuing:

1. *Security-policy specification*: Previous work on security-typed languages has used simplistic ways of specifying information-flow policies. In practice, however, policies are more complex, and, in general, it is not known how to provide a firm theoretical basis for features such as declassification and dynamically changing policies. Figuring out the right way to handle these features is important to understanding the kinds of information security provided by language-based technology.

2. *Integration with encryption*: Encryption is an important mechanism for enforcing information security. While there has been extensive research on the properties of cryptographic systems and protocols that make use of cryptography, relatively little is known about how to take a high-level security policy and express it in terms of appropriate use of encryption primitives. Recent work relating encryption to parametric polymorphism suggests some approaches to this problem, but further study is needed.

3. *Integrity and availability*: While protecting the confidentiality of data has been extensively studied in the context of programming languages, language mechanisms for protecting data integrity and availability have not been thoroughly explored. Replication is a fundamental technique for solving these problems, but there are a variety of implementation choices that offer differing consistency models and costs. Consequently, designing algorithms that make use of replicated data can be difficult. The question is whether the details of replication can be hidden from the programmer by providing suitable abstractions at the language level. Ideally, the programmer would write algorithms that are oblivious to the replication, which the compiler would address. In practice, I expect that some code annotations and static analysis of the program to determine its suitability for replication would be required.

4. *Concurrent programs and information flow*: It is often difficult to reason about concurrent programs, even if they use appropriate synchronization mechanisms, simply because they may exhibit many possible behaviors. Consequently, providing information-flow security in concurrent systems is substantially harder than in the single-threaded case. Existing information-flow techniques for dealing with concurrency often make conservative approximations to the actual behavior of the system, making them too restrictive for practical use. My work on information flow in low-level languages and some recent results by others in this field have suggested some promising directions to look for practical solutions to protecting confidential information in multithreaded and distributed systems.

I am also interested in problems outside the realm of security. I enjoy studying programming-languages theory, and I have recently been intrigued by calculi for concurrent, distributed, and mobile code. Studying these systems may lead to better ways of providing synchronization and locking support for threaded programs. A related question is how to establish (mostly) automatically that multithreaded programs are race free or cannot deadlock.

A more ambitious challenge is to provide programming support for software that runs in ad-hoc networks. What programming models are appropriate for distributed systems in which nodes have limited resources and frequently disconnect from or join the network? How can programmers reason about the behavior of software running in such an environment? What underlying systems support should be provided to these applications? What underlying theory is needed to understand the semantics of these applications?

My approach to these kinds of questions is to formalize the system properties of interest in a programming language. This strategy has the benefit of making the problem precise, and hence amenable to rigorous proof techniques and implementation. However, in my experience, experimenting with new programming language technologies and type systems can be difficult. Establishing that a type system for a programming language is sound is a time consuming and tedious process. Tools for proving type-soundness results and frameworks for experimenting with new languages and the accompanying logics for reasoning about them would benefit not only programming-language researchers, but also the users of the resulting programming languages. The ability to prototype and rapidly create domain-specific type systems and program analyses might be an effective way for software developers to verify intended system invariants or debug existing code.

REFERENCES

1 Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proc. of the 4th ACM SIGPLAN International Conference on Functional Programming*, pages 197–207, Paris, France, September 1999.

2 Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

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

4 Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 2001. To appear.

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

6 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)*, volume 35 of *Operating Systems Review*, pages 1–14. Banff, Canada, October 2001.