Language-based security

Computer security seems like an oxymoron these days, with the Internet providing an easy way to attack computers anywhere in the world. Moreover, virtually all software is designed to be extensible, so you are only a mouse click away from downloading the latest software upgrade or virus —and sometimes it is hard to tell the difference.

The computer security problem has changed dramatically in 40 years. The building-block security properties (confidentiality, integrity, and availability) remain a fundamental part of any solution. The assurance issue also remains crucial —not only must a system be secure, there must be some basis to believe it to be so. But the solution space has decidedly changed due to revolutions in two fields: cryptography and programming languages. Leveraging these developments and further advancing them is the subject of intense activity at Cornell, which has become a leader in a research area that has come to be called "language-based security".

Cornell has become a leader in a research area that has come to be called language-based security.

Forty years ago, the design of programming languages was informed largely by aesthetics and need. A new language design was explored by writing programs for a standard set of problems and writing a compiler so people could use the language. Today, research is focused on program analysis and synthesis agendas, which are applicable to programs in machine-language as well as high-level languages.

- Program analysis methods provide mechanical means to determine whether a program's execution will satisfy certain properties. The properties might be relative to annotations the programmer provides, as in type checking.

- Program synthesis methods provide mechanical means to ensure that execution will satisfy certain properties, by rewriting a program to capture additional state and add additional checks.

Policy enforcement is an obvious target of opportunity. (An example of a policy one might want to enforce is to limit the number of open windows in a GUI, thereby preventing denial-of-service attacks that succeed by opening countless windows.) Prior to executing a machine language program, analysis and synthesis methods can be used to ensure that the program will not violate its policy. CS professor Fred Schneider's work, with student Ulfar Erlingsson, on "in-lined reference monitors" pioneered the idea of using program synthesis to add runtime checks that block execution if a program is about to violate a security policy. Program analysis is then used to delete unnecessary checks prior to execution.

As another example, CS professor Andrew Myers work on enforcing confidentiality and integrity policies employs a mix of synthesis and analysis. Prototyped as an extension of Java, Myers has created an infrastructure that lets programmers use types for defining what hosts in the system are trusted to manipulate the different kinds of data. His system automatically partitions distributed programs into pieces that can safely be executed on each host and generates protocols to coordinate host communication in a way that is consistent with the specified confidentiality and integrity policy.

More surprising than its use in policy enforcement is a role that program analysis and synthesis techniques can play in redefining what constitutes the trusted computing base (TCB) for a system. The smaller the TCB the better, since assurance ultimately comes from people understanding program code, and larger programs are harder to understand. The use of analysis and synthesis techniques in implementing security seemingly adds to the TCB size, but this size increase can be reversed as follows.

An implementation of analysis or synthesis can be instrumented so that it outputs as a formal proof the justification for what that implementation did on a given input program. The formal proof can be bound cryptographically to the input program, resulting in "proof carrying code". The analyzer or synthesizer in a TCB can then be replaced with a proof checker. Proof checking can be implemented by a small, easy-to-understand piece of code. Thus, the replacement reduces the size of the TCB by replacing a relatively large component by a small one. Questions of efficiency —the size of the proof and the cost of checking— and expressiveness remain active areas of investigation; CS professor Dexter Kozen has been exploring these.

The assurance question is being addressed head-on by CS professor Tim Teitelbaum, whose company GrammaTech (founded with former PhD student Thomas Reps) has developed and successfully marketed a collection of tools to help programmers find vulnerabilities and track the flow of information in C, Ada, and C++ programs. These tools are based on static analysis and other program analysis techniques.

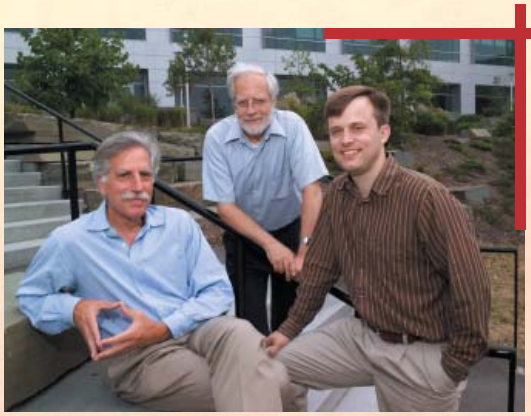
New security defenses lead to new kinds of attacks. Developing specific defenses is important, but keeping ahead of attackers can be unsatisfying because the job is never done. More satisfying is the discovery of general principles about defense mechanisms, and recently this has been aided by applying insights from programming languages. Schneider, for example, has used results from concurrent programming semantics to characterize the class of security policies that can be enforced by in-lined reference monitors. This result not only answered the obvious

Photo: Jon Reis

CS professor Fred Schneider's work, with student Ulfar Erlingsson, on "in-lined reference monitors" pioneered the idea of using program synthesis to add runtime checks that block execution if a program is about to violate a security policy.

question about the new security mechanism but defined a research agenda: characterizing what policies can be enforced by various mechanisms. Other mechanisms have since succumbed: static checkers, program rewriters, and so on.

For those who know CS at Cornell, the style of security work reported here will not be surprising. Our systems work is often tied to principles and often addresses problems that transcend technology or specific engineering issues. "Think first, build second" is a succinct characterization of our primary *modus operandi*, and it continues to serve, as exemplified by the impact the security group is having.



Tim Teitelbaum (left), David Gries (center), and Andrew Myers are part of the Languages and Compilers Group in the department.

Programming methodology and program correctness

The 1968 Nato Software Engineering Conference was a wake-up call for the programming world. For the first time, academicians and industrialists spoke honestly and openly about the software crisis —caused by missed deadlines, massive budget overruns, and software riddled with errors. Everyone admitted they did not know how to program and develop software. The conference inspired research in a number of areas, among them, the correctness of programs.

Tony Hoare's 1969 paper on an axiomatic basis for computer programming provided a foundation for work on correctness by giving a new way to define a programming language —in terms of how to prove a program correct (with respect to a specification) instead of how to execute it.

Cornell got into this field, in terms of education and research, early. The 1973 text *Introduction to Programming*, by CS faculty Dick Conway and David Gries, was the first programming text to take correctness issues seriously, and discussions at Cornell inspired CS professor Bob Constable to begin working on automated proof checking (see p. 36).

Cornell played a significant role in developing approaches to concurrent-program correctness. In 1975, Susan Owicki's PhD thesis, supervised by Gries, provided the fundamental concept of interference freedom; a follow-up paper by Owicki and Gries received the ACM 1977 Programming Systems and Languages Award. Gries used the theory to give one of the first interesting formal proofs, of an on-the-fly garbage collector. CS professor Fred Schneider, Gries, and PhD student Rick Schlichting developed algorithms for fault-tolerant broadcasts, while Schneider, with PhD student Bowen Alpern, developed rigorous definitions of liveness and safety properties and proved that any specification can be decomposed into a safety and a liveness property. Schneider's text *On Concurrent Programming* (1997) provides a comprehensive and rigorous discussion of formal methods for proving concurrent programs correct.

At Cornell, one goal of the work on axiomatic semantics was to learn how a theory of program correctness could influence the programming process —and thus the teaching of programming. Edsger W. Dijkstra, in his monograph *A Discipline of Programming* (1976), gave basic principles and strategies for this. Gries's text *The Science of Programming* (1981) amplified and brought the ideas down to the undergraduate level. There followed a period of intense activity in honing the principles and strategies for developing programs and in developing and presenting algorithms.

Today, correctness issues have not been integrated into the undergraduate computer science curriculum as much as some had hoped. However, these ideas are receiving renewed attention as trustworthy computing initiatives in industry and government turn the spotlight on questions of assurance (see e.g. pp.13-14). Further, formal program development techniques are more and more used routinely in companies concerned with building high-assurance systems. With a formal verification group and all this interest in security, Cornell's faculty will undoubtedly remain leading players in this area.

Tom Reps receives the ACM Doctoral Dissertation Award for his PhD thesis, *Generating Language-Based Environments* (MIT Press). Reps, whose advisor was Tim Teitelbaum, is now a Professor at Wisconsin, Madison.

Gianfranco Bilardi, Alexandru Nicolau, John Solworth, Vijay Vazirani join.

1984

The Synthesizer Generator is distributed to over 330 institutions. Developed by Tim Teitelbaum and student Tom Reps, this tool for automating the construction of interactive language-based environments is based on Reps's 1983 thesis prototype. The Synthesizer Generator was subsequently commercialized and is still in use.

Gene Golub and Charlie Van Loan publish *Matrix Computation* (Johns Hopkins Press).

The CS computing facility serves as the gateway for the entire university to Arpanet and CSnet. CS is instrumental in the university's Project Ezra to increase the use of computers on campus, with a 5-year, $8 million grant from IBM.

Prakash Panangaden, Dexter Kozen join.

1985

The Cornell Theory Center, founded in 1984, becomes one of four NSF supercomputer centers. IBM provides an additional $30 million in hardware, software, and staff.

Ken Birman develops the first version of Isis, the first system for fault-tolerance in distributed systems. Isis has impacted the theory and practice of distributed computing. Two years later, the virtual synchrony model is defined and incorporated.

CS receives its second 5-year NSF CER (Coordinated Experimental Computing) grant.

David Gries receives the AFIPS Education Award for his contributions to computer science education.

Keith Marzullo, Alberto Segre, Keshav Pingali join.

1986

The Nuprl work reaches a milestone: Bob Constable and his students publish *Implementing Mathematics with the Nuprl Proof Development System* (Prentice Hall).

John Hopcroft shares the ACM Turing Prize with Bob Tarjan, "For fundamental achievements in the design and analysis of algorithms and data structures". The work was Bob Tarjan's PhD thesis at Stanford, advised by Hopcroft. Their major achievement was a linear algorithm for graph planarity testing, but many more ideas on algorithm design and data structures came out of their collaboration.