Steve Zdancewic
Research Statement
September 4, 2007

Security-Oriented Languages

The proliferation of network-enabled devices and the success of Internet applications has made it easier to access data and share computing resources, improving many aspects of day-to-day life. At the same time, our society has become dependent on computing infrastructure for education, entertainment, business, the military and government transactions. As a consequence, ensuring the security and reliability software systems is a crucial and challenging problem that has the potential to impact society at all levels [20].

Programming language approaches have emerged as a promising technique to improve both software reliability and security. (See, Schneider, Morrisett and Harper's summary paper [21].) Two key ideas in this domain are *static program analysis* (often presented as type systems), in which the compiler validates desired properties of the source program, and *dynamic checks*, in which the executable code is instrumented (by hand or automatically) with additional instructions that determine at run time that the program behavior is permissible. These complementary mechanisms can enforce a variety of desirable, security-relevant properties such as memory safety, access control, and information-flow constraints. The success of this approach is seen in modern programming languages such as Java and C#, which employ both strategies.

Despite this progress, it is still difficult to build secure software systems using Java or C#. Part of the problem is that these languages do not provide the right abstractions for describing appropriate security policies. For example, although the Java libraries provide cryptographic primitives for encryption and digital signatures, it is difficult to ensure that private keys or other confidential data are kept secret. Another difficulty is ensuring that appropriate authentication and access control checks are performed before allowing the program to execute potentially dangerous operations. The goal of my research is to apply the traditional techniques of programming language technology (semantics, type systems, compilers, etc.) to these types of problems, thereby improving software security and reliability. For the past several years, my research has primarily concentrated on protecting the confidentiality and integrity of data in software systems by enforcing *information-flow* security policies.

Information-flow security in programming languages

Languages that support information-flow security constrain how programs handle confidential or potentially untrustworthy data to ensure that it does not leak to inappropriate locations [19]. The correctness of such a language is usually defined by a property called *noninterference* [6], which requires that secret data not affect publicly observable behavior of the program. In practice, such a definition of security is too restrictive, because many programs either are intended to leak some secret information or have policies that evolve over time. Since joining Penn I have focused primarily on four (somewhat intertwined) aspects of language-based information-flow security: (1) theoretical underpinnings, (2) downgrading, (3) dynamic security policies, and (4) developing applications of these techniques. This research has been supported through NSF grants via the CAREER and Cybertrust programs.

Information-flow policies One thread of my research is studying the theoretical foundations of information-flow policies, how various notions of security relate to one another, and how information-flow properties relate to other concepts.

One part of this work, conducted with my graduate student Peng Li, has investigated the relationships between confidentiality and integrity properties [8, 11]. We conclude that, contrary to the conventional wisdom (and standard

practice within parts of the formal security community), integrity and confidentiality should not be treated as pure duals to one another. This result suggests that information-flow policies for integrity deserve more attention in the community than they currently receive.

With Rajeev Alur and his student Pavol Černý, I have investigated the connections between information-flow properties and those expressible using standard specification logics for model checking. We show that a very general, but natural, definition of information-flow is not definable in mu-calculus, and thus cannot be checked using standard model checkers. However, existing tools based on refinement checking can show correctness of an implementation with respect to a specification. This work appeared in the 2006 International Colloquium on Automata, Languages and Programming (ICALP) [2]

Also along these foundational lines, my graduate student, Stephen Tse, and I demonstrated that the noninterference property guaranteed by information-flow languages can be enforced using *parametric polymorphism*, a property well known in the functional programming languages community. This result, which appeared at the 2004 International Conference on Functional Programming (ICFP) [24] has practical significance because it opens up the possibility of enforcing strong information-flow policies in existing languages that already have support for polymorphism, such as Haskell, ML, Java 5.0 or C#. An extended version of this paper has been accepted to the Journal of Functional Programming, pending revisions.

Downgrading: Another thread of my research is studying downgrading mechanisms. Downgrading policies allow programs to convert secret information into public information—for confidentiality, this process is called *declassification*; for integrity it is called *endorsement*. Downgrading in some form is necessary in many situations. For example, to implement a secure login prompt requires declassification of the outcome of comparing the user-supplied input with the corresponding entry in the password database. Whether or not the login attempt is successful reveals a small amount of information about the password database.

Clearly, such downgrading, though useful in practice, is potentially dangerous and thus must be handled carefully. My work attempts to permit information-flow policies that include useful downgrading mechanisms, while still maintaining the ability to reason about the consequences of using a particular policy.

One approach to limiting the use of downgrading mechanisms is to associate *authority* with the code and then check statically that the program has enough authority to perform the downgrading it contains. This alone is not robust in the presence of untrusted information, which may cause unintentional release of secrets via downgrading operations. To improve this situation, I and my colleagues Andrew Myers (at Cornell University) and Andrei Sabelfeld (at Chalmers University, Sweden) developed a notion of *robust declassification*. Intuitively, robustness requires that the decision to perform a sensitive downgrading action must be trusted by any principal whose information-flow constraints are weakened by the downgrading. This line of research has been presented at the conference on Mathematical Foundations of Program Semantics (MFPS) [28], the Computer Security Foundations Workshop[1] (CSFW) [16, 29], and in the Journal of Computer Security [17].

With my student, Peng Li, I have also investigated an alternative, more semantic approach to downgrading policies. The idea is to allow the programmer to specify under what circumstances a secret may be considered public or when tainted data can be considered untainted. For instance, while an the entire credit-card number may be secret, the last four digits may be considered public. The beauty of this approach is that it leads to a natural, extensional relaxation of the standard definition of noninterference; we presented this work in a paper that appeared in the 2005 Symposium on Principles of Programming Languages (POPL) [9].

More recently, I have also studied the interaction between cryptography and information-flow policies. This work, done with my student Jeffrey Vaughan, has resulted in a programming model that allows software developers to specify confidentiality and integrity requirements abstractly—the compiler inserts appropriate uses of encryption and digital signatures when data enters or leaves the system. With this approach, programmers are freed from the burden of having to reason about cryptography and explicit key management; the compiler ensures that the software meets its information-flow policy. This work was presented at the 2007 IEEE Symposium on Security and Privacy [26].

[1] Despite the word "Workshop" in its title, CSFW is a competitive publication venue that accepts approximately 20-25 papers out of 100 or more submissions. CSFW has existed for 20 years and in 2007 was raised to 'Symposium' status by the IEEE.

Dynamic policies: A third thread of my research connects the static analysis in a language's type system with dynamic information (such as which user ran the program, or what the current OS-determined access control permissions are for a given file). Dynamic policies are useful for integrating language-based mechanisms with more traditional mechanisms like those provided by the OS. My student, Stephen Tse, and I have studied the properties of these dynamic policies and showed how one instance of them can be soundly incorporated into standard information-flow type systems. The idea is to allow programs to use *run-time principals*, which are first-class data objects representing users, groups, etc. During its execution, a program may inspect a run-time principal to determine policy information not available when the program was compiled. The key problem is designing a language such that the dynamic checks needed to implement run-time principals introduce no additional covert channels; moreover the new dynamic policies should interact well with purely static policies, including those with downgrading. Run-time principals provide a means of integrating policies expressed by the type system with external notions of principals such as those provided by the OS or a public key infrastructure. This work was presented at the 2004 IEEE Symposium on Security and Privacy [23] and an extended version will appear in Transactions on Programming Languages and Systems (TOPLAS) [25].

Another aspect of dynamic policies is accommodating changes to policy while the system is running. For example, a system administrator might want to revoke or augment privileges for an employee whose role within a company has changed. This problem has been well studied for access control policies, but had not been studied in the context of information-flow policies. Working with Michael Hicks (at the University of Maryland, College Park) and graduate students Boniface Hicks, Nikhil Swamy and Stephen Tse, I have developed techniques that use software memory transactions to allow information-flow policy updates in long-running software such as servers or operating systems. This work appears in FCS [7] and CSFW [22].

Applications: The fourth thread of my research seeks to apply information-flow techniques in useful settings. Based on our earlier, more theoretical work, Peng Li and I demonstrated one potentially practical application of our relaxed noninterference techniques for web security in a CSFW paper [10]. We developed a domain-specific web scripting language intended for interfacing with databases. The language allows safe downgrading according to programmer specified policies. This novel, pattern-based approach provides a machine-checked way of enforcing data integrity policies that are often implemented in ad-hoc ways.

Also with Peng Li, I have shown how to embed a domain-specific security-oriented language in the programming language Haskell [18]. The sublanguage provides useful information-flow control mechanisms including dynamic security lattices, run-time code privileges and declassification, without modifying the base language. Our design avoids the redundant work of producing new languages, lowers the threshold for adopting security-typed languages, and also provides great flexibility and modularity for using security-policy frameworks. Using this approach, we have shown how it is possible to implement software that makes strong information-flow guarantees using existing language technology. This work appeared in CSFW 2006 [12].

Another application of language-based security is my work with Ph.D. student Karl Mazurak. We have designed and implemented a tool for statically analyzing programs written in the `bash` scripting language. Although it makes no formal guarantees against missed errors or spurious warnings (largely due to the highly dynamic nature of bash scripts), this tool is useful for detecting certain common program errors that may lead to security vulnerabilities. In experiments with 49 bash scripts taken from popular Internet repositories, we were able to identify 20 of them as containing bugs of varying severity. This work appeared in the 2007 SIGPLAN workshop on Programming Languages and Analysis for Security [15].

Additional research

Besides my primary focus on security, I am also interested in a wide range of other programming-language topics. Two significant non-security projects I have worked on at Penn are described in this section.

Haskell concurrency library: My work with Peng Li on embedding secure languages into Haskell led us to the question of whether secure concurrent programs could be built in the same way. Before exploring the security aspects

of concurrency in Haskell, we first needed to understand how concurrency alone could be treated. Building on earlier work by Koen Claessen [5], we developed an approach for building application-level concurrency using *concurrency monads*, which provides type-safe abstractions for both events and threads. This approach simplifies the development of massively concurrent software in a way that scales to real-world network services. Our Haskell implementation supports exceptions, symmetrical multiprocessing, software transactional memory, asynchronous I/O mechanisms and application-level network protocol stacks. Experimental results demonstrate that this monad-based approach has good performance: the threads are extremely lightweight (scaling to ten million threads), and the I/O performance compares favorably to that of Linux NPTL. This work was published at the 2007 conference on Programming Language Design and Implementation (PLDI) [13].

A theory of Aspect-Oriented Programming: Along with David Walker (of Princeton University) and his graduate student Jay Ligatti, I developed a core calculus to give an operational explanation of aspect-oriented programs. Our approach is to give a type-directed translation from a user-friendly external language to a compact, well-defined core language. We argue that our framework is an effective way to give semantics to aspect-oriented programming languages in general because the translation eliminates shallow syntactic differences between related constructs and permits definition of a simple and elegant core language. This work appeared first in the International Conference on Functional Programming (ICFP) [27] and then later in the Journal of Science of Computer Programming [14].

Community impact

POPLmark: One significant obstacle to verified software is reasoning about the languages in which the software is written. Without formal models of programming languages, it is impossible to even state, let alone prove, meaningful properties of software or tools such as compilers and type systems. The programming language research community therefore relies on models of programming languages that are used to prove *metatheoretic* properties—properties about the language itself—by hand. Unfortunately, many proofs about programming languages are straightforward, extremely long, and tedious, with just a few interesting cases. Their complexity arises from the management of many details rather than from deep conceptual difficulties; yet small mistakes or overlooked cases can invalidate large amounts of work. These effects are amplified as languages scale: it becomes very hard to keep definitions and proofs consistent, to reuse work, and to ensure tight relationships between theory and implementations. Automated proof assistants offer the hope of significantly easing these problems, however, despite much encouraging progress in recent years and the availability of several mature tools, their use is still not commonplace.

Benjamin Pierce, Peter Sewell (of Cambridge University), Stephanie Weirich, our graduate students, and I have collaborated to improve this situation. In 2005 we issued the *POPLmark Challenge* [4], a competition that urged the programming languages community to test out mechanized proof assistants for conducting their research. The response has been quite dramatic: our efforts spawned a mailing list of over 150 researchers, we established a new Workshop on Mechanized Metatheory that had more than 40 attendees for its first meeting (its second meeting will take place in September 2007), researchers have made great progress in using the Coq, Isabelle/HOL, and Twelf tools for conducting language research, and we have several complete solutions to the POPLmark challenge. We are now in the process of creating tutorial material and library implementations for doing metatheory in Coq. Our tutorial will be given at POPL 2008 and we have plans to turn the tutorial material into a textbook.

PLAS: Programming Languages and Analysis for Security Although the idea of improving computer security by using programming language techniques has been around for many years, until recently there has not been a forum dedicated solely to this topic. Instead, researchers have published in a variety of conferences, each focused on a different aspect of the area, either programming languages (ICFP, POPL, PLDI, etc.), systems (OSDI, SOSP, etc.) or security (CCS, Oakland, Usenix Security, CSF, etc.). Recognizing that the community would benefit from having a single meeting to bring together researchers in programming-languages based security, in 2005, Vugranam Sreedhar of IBM Research, T.J. Watson, and I founded the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS). Although still in its infancy, support for the workshop has been strong—there were almost 40 attendees at the first meeting in 2006 and more at the second meeting in 2007. The number of submitted papers

doubled from 18 in the first meeting to almost 40 in the second instance; in each case only about 10 were accepted for presentation during the workshop. Preparations are underway for the third instance of the meeting, to be held in conjunction with PLDI in 2008.

Ongoing and future work

Implementation, access control, and auditing: My current research plan is to continue the language-based security projects described above, extending them in two directions. First, although my research group has built a number of prototype implementations to test out our designs and examples, I feel that it is now time to concentrate on a full-scale language implementation. To that end, we are currently designing and implementing a compiler intermediate representation and type system amenable for expressing strong information-flow security properties. Our design encompasses many of the concepts mentioned above, including dynamic security policies and robust downgrading mechanisms. This implementation will serve both as a larger scale test of the concepts and as a platform for conducting future research.

My second current research direction, which will be integrated with the compiler implementation described above, is to combine access-control mechanisms in the style of proof-carrying authentication [3] with information-flow policies. This idea originates from a surprising observation made by Martín Abadi [1] that a certain type system for defining information-flow properties coincides with a very natural authorization logic, which is useful for expressing decentralized authorization proofs in the style of well-known trust management mechanisms. My hypothesis is that the resulting policy language will be a natural way of specifying information-flow policies qualified by access-control requirements. I also project that this approach will also provide a strong foundation for program auditing—there has been surprisingly little study of software auditing policies and mechanisms.

Hardware support for safe languages: With Milo Martin, I am also investigating how hardware can help improve the performance, ease of use, and backwards compatibility of safe programming languages. Our current work focuses on providing hardware primitives that enable simple and complete array-bounds checks for low-level programming languages like C. Inspired by the promise of software-only approaches, this research proposes a *hardware bounded pointer* architectural primitive that supports cooperative hardware/software enforcement of spatial memory safety for C programs. This bounded pointer is a new hardware primitive datatype for pointers that leaves the standard C pointer representation intact, but augments it with bounds information maintained separately and invisibly by the hardware. The bounds are initialized by the software, and they are then propagated and enforced transparently by the hardware, which automatically checks a pointer's bounds before it is dereferenced. When combined with simple intra-procedural compiler instrumentation, hardware bounded pointers enable a low-overhead approach for enforcing complete spatial memory safety in unmodified C programs.

References

[1] Martín Abadi. Access control in a core calculus of dependency. In *Proc. 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 263–273, 2006.

[2] Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *Proc. of 33rd International Colloquium on Automata, Languages and Programming*, 2006.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proc. of 6th ACM Conference on Computer and Communications Security*, November 1999.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The POPLMark Challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2005.

[5] Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[6] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[7] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

[8] Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, September 2003.

[9] Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 158–170, January 2005.

[10] Peng Li and Steve Zdancewic. Practical Information-flow Control in Web-based Information Systems. In *Proc. of 18th IEEE Computer Security Foundations Workshop*, pages 2–15, 2005.

[11] Peng Li and Steve Zdancewic. Unifying Confidentiality and Integrity in Downgrading Policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

[12] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proc. of 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[13] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services. In *Proc. 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2007.

[14] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming*, 2006.

[15] Karl Mazurak and Steve Zdancewic. ABash: Finding bugs in bash scripts. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2007.

[16] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Asilomar, CA, June 2004. IEEE Computer Society Press.

[17] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[18] John Peterson, Kevin Hammond, Lennart Augustsson, et al. Report on the programming language Haskell, version 1.4. http://www.haskell.org/report.

[19] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[20] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.

[21] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary.*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101, Saarbrücken, Germany, August 2000. Springer-Verlag.

[22] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proc. of 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[23] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*. IEEE Computer Society Press, May 2004.

[24] Stephen Tse and Steve Zdancewic. Translating Dependency into Parametricity. In *Proc. of the 9th ACM SIGPLAN International Conference on Functional Programming*, 2004.

[25] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *Transactions on Programming Languages and Systems*, 2006. to appear.

[26] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE 2007 Symposium on Security and Privacy*, 2007.

[27] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. In *Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming*, Upsala, Sweden, August 2003.

[28] Steve Zdancewic. A Type System for Robust Declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science, March 2003.

[29] 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.