Research Statement

Nikhil Swamy

The overarching goal of my research is to improve the quality of software, primarily by ensuring that it is secure. By designing new programming languages, novel program analyses and language runtime environments, I aim to make software secure *by construction.* My dissertation work has focused on securing multi-tier web applications. To that end, I have designed and implemented *SE*LINKS, a language in which a wide class of security policies can be reliably enforced end-to-end. I have built two realistic applications in this language and my experience illustrates that *SE*LINKS makes it easy to enforce common policies, like access control, while allowing more complex policies, like information flow, to be enforced with considerably greater effort on the part of the programmer. This characteristic has been a guiding principle of much of my research and I expect it will inform my future work too. It is summed up by the following two propositions:

- Many common programming tasks, though conceptually straightforward, are easy to get wrong. Languages should be able to protect against common errors with little programmer effort.
- Most interesting program properties are very difficult to establish without assistance from the programmer. Languages (or analysis tools) should be expressive enough to allow programmers to state and prove complex invariants for code that is considered critical. The guarantees on a program's behavior should be commensurate with the additional effort invested by the programmer.

I first came to appreciate this principle when working on extensions to the Cyclone programming language [25]. Programming in Cyclone is straightforward if one is content having all dereferences bounds checked at runtime and relying on the garbage collector to manage memory. However, with considerable additional effort in terms of annotating and restructuring a program, Cyclone allows the programmer to statically prove many array accesses to be within bounds and to precisely and safely control memory usage by explicitly deallocating objects.

In the remainder of this statement, I present a selection of recent research projects on which I have been one of the main contributors. Together, these projects address important, practical, and diverse security concerns of real web applications. In each case, I summarize the main novelty of the work and discuss how it fits in with my overall research vision. I also discuss ideas for future work which I expect will form the basis of my research agenda in the coming years.

1 *SE*LINKS: Enforcing User-defined Security Policies in a Web Application

Most large organizations must maintain a substantial information presence on the world wide web in order to share information with their partners and customers. Yet it is their web-facing resources that are the most vulnerable to security breaches. A methodology by which to make web applications secure by construction stands to significantly reduce the tension between the need to share information widely and the threat of critical losses due to information leaks.

It is exactly the promise of security by construction that security-typed programming languages bring. First proposed by Volpano, Smith, and Irvine [26], security-typed programming languages allow information flow policies to be verifiably enforced by type checking. While this field has received considerable attention in the literature [20], to date only two programming languages implementations, Jif [8] and FlowCaml [23], have incorporated elements of the theory of security-typing, with Jif the more practical. However, programming in Jif can be an onerous task, and very few realistic applications have ever been constructed [12]. The principal difficulty arises from having to account for many potential forms of information leakage in an effort to enforce *noninterference*, a theoretically-pleasing security property, but one which is often too strong for real programs.

The main insight behind my dissertation work, under the guidance of my advisor Michael Hicks, is that to fulfill its promise, security-typing must allow many common security policies (like access control) to be *correctly* and *easily* enforced in a program. If more restrictive policies are required for some critical components of an application, then these policies should still be enforceable, although this may require considerably more programmer effort.

With these goals in mind, I have designed FABLE, a core formalism for enforcing user-defined security policies [24]. By using a simple form of dependent typing [2], I have shown that a range of security policies (including access control, static and dynamic information flow and provenance tracking [5]) can be correctly enforced in a program. When substructural types [27] are added to the system, an even wider range of policies (including stack inspection [11] and security automaton policies [22]) can also be provably enforced. The basic idea is that policy designers can define a language of security labels and associate a label with each protected resource in an application. The semantics of these labels is given by snippets of *policy code*. The type system ensures that the association between data and their labels is maintained consistently throughout the program and that all operations on labeled data are mediated by the policy code.

I put FABLE to use by extending the type checker for the LINKS programming language [9] to include the checking of FABLE-style dependent types; I call the resulting language *Security-Enhanced* LINKS, or *SE*LINKS. LINKS is a language for programming multi-tier web applications—applications that include client code typically running as JavaScript in a web browser; web server code that implements much of the application logic; and data access code that interfaces with a persistent store. It provides a uniform set of abstractions for programming each of these tiers and the LINKS compiler generates code, say, SQL or JavaScript, that is best suited to running on a given tier of the application. This feature of LINKS has made it possible to focus on the key security and language related issues without getting sidetracked by the engineering challenges presented by the multilingual frameworks that are commonly used to construct commodity web applications. Whereas promising recent advances such as Swift [7] allow information flow policies to be enforced for Java servlet applications, I anticipate applying the lessons learned from *SE*LINKS to enforce a wider range of policies, with varying degrees of ease of use, for web applications constructed in mainstream frameworks like JavaScript/J2EE or JavaScript/PHP/MySQL.

To evaluate *SE*LINKS another student and I have used it to build two relatively large, secure, web applications. One application is a secure document management system, in the style of a wiki or a blog, that allows fine grained access control and data provenance policies to be associated with the fragments of a structured document [10]. The second application, a port of an e-commerce application that ships with LINKS, allows label-based access control policies to be associated with a customer's order information. Our initial experience indicates that *SE*LINKS makes it is easy to enforce policies, like access control, that govern only direct data flows. Recently, I have begun exploring some ways to reduce the burden of enforcing more complex policies by automatically transforming a program according to a programmer-defined set of rewriting rules.

1.1 Future Work

My long-term goal is to develop a tunable framework for security policy enforcement. That is, one should be able to customize the security behavior of an application by "plugging in" a high-level policy and its corresponding enforcement mechanism in a manner that best suits the particular needs of that application's deployment. I envisage a number of specific research projects to extend my work on *SE*LINKS that makes progress towards this goal either by enhancing the expressivity of policy enforcement idioms or by making it easier to reason about and enforce more complex policies.

Policy composition. The security model that *SE*LINKS provides is almost entirely user-definable. As such, it is common, even in our example applications, for very different security policies to govern different parts of the same application. However, more interesting compositions of policies are also natural. For instance, a policy might state that data governed by *Alice*'s access control policy is subject to a lattice-based information flow policy once it is released to *Bob*. While the enforcement of *Alice*'s access control policy and the lattice-based policy may have been proved correct in isolation, it is not immediately clear that the composed policy does not violate the invariants of its components, much less that it achieves the desired composed semantics.

One interesting idea here is the notion of a policy module that, in the style of module systems for programming languages like ML, clearly defines all the invariants that must be preserved for a policy to be composed with other policies. A promising starting point for this research would be to leverage existing work into noninterference-based policies. For instance, if it can be shown that one policy never inspects or influences data that another policy depends on (and vice versa), then policies can be safely composed while preserving the invariants established when reasoning about each policy in isolation. For policies that are intended to have some form of interaction, more elaborate conditions would have to be developed. It is possible that the growing literature in downgrading policies [21] can be brought to bear here.

Multi-tier policy enforcement. Application-level policy enforcement in *SE*LINKS is attractive in many ways. For instance, multiple applications that access a database may each enforce a security policy that best suits its needs. However, enforcing a security policy within the database can be useful for many reasons too. One may wish to enforce a common, base-level security policy across all applications. Additionally, policies that apply to bulk data can be enforced much more efficiently within the database. One direction of research that I have begun to explore is a compilation strategy for security policy code to PL/SQL so that it can run independently of a particular *SE*LINKS application, within the database. Likewise, being able to run policy code within the client tier is also attractive for several reasons. However, this is somewhat more difficult since the client is inherently untrustworthy. One approach to this problem is outlined in Section 2.1.

Partial-mechanization of Fable metatheory. At present, proving that a user-defined FABLE policy correctly enforces some high-level security property is entirely manual. My experience with proving the correctness of several policy encodings indicates that these proofs, although non-trivial, are simplified considerably by the type-soundness results that I have proved for FABLE. Following a proposal by Millstein et al. for partially automating the proofs of correctness of user-defined type extensions [15], this project would attempt to translate FABLE policy code to definitions in a proof assistant, like Coq [3]. Whereas Millstein et al. attempt to prove relatively simple syntactic properties, proving semantic properties like noninterference would require a substantially larger effort. However, given my experience with manual proofs, I conjecture that when given a Coq library that formalizes the metatheory of FABLE, the policy analyst could rely on several key FABLE soundness lemmas to discharge a proof of a security property. Such a facility would be particularly useful within a tunable policy enforcement framework—not only could a policy be reliably enforced in the program but an extensional security property about the program and policy could also be proved with a little more effort.

2 BEEP: Defeating Cross-site Scripting Attacks

A cross-site scripting (XSS) attack can occur when a web page contains script content from a third party, such as an advertiser or other users. This script executes in the web browser with the same level of privilege as scripts that originated from the server and can steal steal private information from the client's web browser and possibly co-opt the client's web browser into attacking other web sites. XSS attacks were identified as the most common security vulnerability in 2007 [16].

A common defense against XSS has been to process web pages at the server to filter out any malicious script-like content. However, reliably identifying script content from standard HTML is highly error-prone. The reason is that in order to cope with the high variability of web pages, browsers typically are willing to parse wildly malformed pages, with each browser typically accepting a different kind of ill-formed HTML. A perfect server-side filter would have to mimic the behavior of all browsers, a virtually impossible task. Other forms of traditional language-based analyses, such as preventing user-tainted data from being included in a web page, are largely inapplicable in this setting. Web sites *want* their users to be able to include rich content in their web pages, and although much of that content is harmless, some of it is malicious. Simply treating all user-provided data as tainted is unacceptable.

Trevor Jim, Michael Hicks, and I designed a practical and arguably perfect solution to this problem. The solution, termed browser-enforced embedded policies, or BEEP [14], relies on our insight, obvious in hindsight, that the JavaScript interpreter in every web browser is the last line of defense against XSS attacks. If we can reliably identify the malicious scripts in the web browser, then it becomes possible to filter them just prior to execution by the JavaScript interpreter.

Distinguishing the good scripts from the bad is still a hard problem. Approaches such as BrowserShield [19] attempt to solve this problem automatically by rewriting scripts at a web proxy. The rewritten scripts must access protected resources through trusted code. But this approach is unappealing mainly because the the rewriting policy is expected to be specified independently of the site that serves the web page. This makes it difficult to distinguish between a trusted server script that legitimately accesses some cookie in a browser from a malicious script that does the same. In BEEP, we enlist the programmer to assist with the detection of the malicious scripts. The many web pages that constitute a complex web application are typically generated from some form of template that embeds dynamic content (which may contain malicious scripts) within static content. Thus, it is relatively easy for a web application developer to add annotations to a web page that identifies either the good scripts or the bad ones.

We explored two kinds of annotations. In the first approach, the web developer specifies a *whitelist* of scripts by including a cryptographic digest of the each script that is legitimately required for the proper functioning of the application. Prior to loading a script, the JavaScript interpreter computes a digest for the script and checks it against the whitelist. We provide simple tool support to allow these whitelists to be constructed from the static content of a page. A dual *blacklist* approach is for the developer to structure a page to sandbox all potentially malicious content within a particular part of the page. The JavaScript interpreter will refuse to run any script content found in the sandbox. We demonstrated the effectiveness of our solution by showing that our BEEP-enabled versions of several open source browsers successfully detected and rejected all scripts in a large corpus of malicious scripts and HTML found in the wild.

2.1 Future Work

Enhancing client-side monitoring. As Web 2.0 applications with very rich client side features, continue to gain prominence, enriching a browser's JavaScript runtime environment with the ability to enforce complex policies will become increasingly important. Java applets, which were once the main vehicle of interactive content on web pages, have been supplanted by AJAX-enabled JavaScript. Following the example of Java, where expressive security policies ranging from sandboxing to stack inspection were found to be necessary, one might expect that it will be useful to enforce non-trivial JavaScript security policies within the browser. The policies currently implemented in BEEP are particularly simple. A script is either trusted or it is not. Once a trusted script is permitted to execute, it is free to do whatever it pleases, including load other scripts. An interesting direction for future work would be to equip JavaScript interpreters with the ability to perform BrowserShield-style rewriting using policies specified, as in BEEP, on a site-specific basis.

Server-side monitoring of application invariants. BEEP protects clients from malicious code that might be served with a web page. However, a server also needs to be protected from a malicious client. For instance, web applications have complex control-flow properties that govern a client's workflow through the application. Violations of this workflow can cause the web application to enter an inconsistent state.

One approach to solving this problem is to use a FABLE-style security automaton policy to ensure that each client request is consistent with the current state in the workflow of an application. We have experimented with this enforcement technique in our *SE*LINKS e-commerce application. However, other techniques may also be applicable. For instance, ideas from system-call monitoring [13], originally developed to ensure that an operating system's integrity is not compromised when an application is attacked, can also be applied to web applications. A server side monitor could intercept all client requests and ensure that they conform to some specification of the client's expected behavior. Additionally, one might be able to adapt ideas from kernel-based control-flow integrity monitors to suit web applications [18]. By analyzing the source of a web application, one could automatically extract a model of the client's behavior which could then be enforced by a server-side monitor, on a per-session basis. Such an approach could be particularly effective for a *SE*LINKS program, in which the entire application's source could be analyzed at once to extract a precise model of its expected control-flow behavior.

3 Verified Implementations of Cryptographic Protocols

In today's information economy, digital identity has begun to replace more traditional forms of identity such as business cards. Federated identity-management systems [1, 17] allow trust relationships based on these identities to be communicated, thereby providing a web of trust on which secure transactions can be conducted. These technologies are beginning to reach critical mass—Windows Vista now ships with the CardSpace system that layers federated identity-management features on top of a suite of web services protocols. As these technologies become pervasive, we must be able to ensure both the privacy of individuals and the security of online transactions that rely on these protocols. In joint work with Karthik Bhargavan, Cédric Fournet and Andy Gordon during my internships at Microsoft Research, I worked on extending and applying a suite of tools to verify *implementations* of the Information Card family of protocols, the underlying specification of Windows CardSpace [4].

Our approach involved first constructing *reference implementations* of the protocols. A reference implementation is one that is fully compliant with the protocol specification, but is implemented in a manner optimized for clarity, rather than for performance. Next, we compiled these implementations to scripts in the language of an automated theorem prover, specialized towards proving cryptographic properties of communicating processes. Designing the reference implementations in a manner that allows the theorem prover to discover a proof requires much discipline and effort on the part of the programmer. It is a subtle, iterative process, reminiscent of restructuring and annotating code to prove that the invariants dictated by a program analysis are true. But, the return on this investment for the programmer is correspondingly large. We were able to automatically prove deep properties of our reference implementations, corresponding to secrecy and authenticity properties for the underlying protocols. Our results include the verification of the largest multi-party protocols implementations, to date.

By ensuring that Windows CardSpace implementations of the protocols interoperate with our reference implementations we increase confidence in the correctness of the production code. Furthermore, our reference implementations provide an executable version of the Information Card specification, which allows other vendors to easily test compliance with a complicated specification. Finally, we were able to automatically generate variants of the reference implementations from metadata that gives values to the several configuration parameters that are available in Information Card. This opens the possibility of verification *on demand* for protocol configurations—one can choose to interact with a web service only if the configuration metadata published by the service produces a client implementation that satisfies the desired security goals.

3.1 Future Work

Policy composition. The family of protocols defined by the web services security policy specifications are designed in a manner that permits them to be composed in multiple ways. For instance, multi-party protocols can be chained together almost arbitrarily, and protocols can be layered on top of each other. Additionally, each protocol is customized by a large number of parameters. As with FABLE policies, a major challenge here is proving that policy customizations and compositions satisfy desired security properties. One possible approach here is to begin to apply ideas from universally composable security properties to web services protocols [6]. By showing that, under certain assumptions, a component of a cryptographic protocol is indistinguishable from an idealized process that establishes the security properties of that component alone, we can hope to recover a degree of modular reasoning about the composed policies. To be feasible, this approach will require a less precise model of cryptography and will likely yield a weaker overall theorem. However, this may be an acceptable cost to pay in order for formal methods to scale to policies with complex compositions.

Conclusion

I have brought my background in formal methods and my expertise in constructing software to solve a range of real software problems. The three projects I have described here all share the common thread of improving various aspects of security for web applications. I have also worked on numerous other projects, ranging from ensuring the memory safety of device drivers by programming them in Cyclone to detecting race conditions in Java programs. The selection of future projects I have outlined here aims to extend my prior work both in ambitious directions, like policy composition and mechanized correctness proofs, as well as in well-defined shorter term projects, like enhanced client-side monitoring and cross-tier policy enforcement. Perhaps more important is my willingness and ability to tackle practical problems that affect software. No doubt, these problems will continue to serve as a consistent source of interesting and challenging problems for some time to come.

References

[1] Liberty alliance. `www.projectliberty.org`.

[2] D. Aspinall and M. Hoffmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT Press, 2004.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] K. Bhargavan, C. Fournet, A. D.Gordon, and N. Swamy. Verified implementations of the information card federated identity-management protocol. In *ACM Symposium on Information, Communication and Comunication Security (ASIACCS)*, 2008.

[5] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[6] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[7] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[8] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2006.

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. 2006.

[10] B. J. Corcoran, N. Swamy, and M. Hicks. Combining provenance and security policies in a web-based document management system. In *In Online Proceedings of the Principles of Provenance Workshop*, 2007.

[11] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[12] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[13] T. Jim. System call monitoring using authenticated system calls. *IEEE Trans. Dependable Secur. Comput.*, 3(3):216–229, 2006. Member-Mohan Rajagopalan and Member-Matti A. Hiltunen and Fellow-Richard D. Schlichting.

[14] T. Jim, N. Swamy, and M. Hicks. Defeating scripting attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 601–610, May 2007.

[15] D. Marino, B. Chin, T. Millstein, G. Tan, R. J. Simmons, and D. Walker. Mechanized metatheory for user-defined type extensions. In *ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[16] MITRE. Common vulnerabilities and exposures. `http://cve.mitre.org`.

[17] A. Nanda. *A Technical Reference for the Information Card Profile V1.0.* Microsoft Corporation, December 2006. At `http://go.microsoft.com/fwlink/?LinkId=87444`.

[18] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, Oct. 2007.

[19] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

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

[21] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2005.

[22] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[23] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *APPSEM-II*, pages 152–165, Mar. 2003.

[24] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *29th IEEE Symposium on Security and Privacy*, 2008.

[25] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in Cyclone. *Science of Computer Programming (SCP)*, 62(2):122–144, Oct. 2006. Special issue on memory management.

[26] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[27] D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2004.