# Research Statement

Deian Stefan

My research interests span the areas of systems, programming languages, and security. I particularly enjoy building secure systems that can see adoption. My efforts are generally guided by two goals: (1) to enable average developers to build secure systems and applications, and (2) to leverage the benefits of formal semantics when reasoning about the security properties of a system. For example, as part of my thesis research, I built a framework (Hails [6]) that allows novice developers to build secure web applications. I then implemented a browser security architecture (Confinement with Origin Web Labels, or COWL [19]), currently being standardized at the W3C [20], for protecting user privacy from untrusted JavaScript. For both systems, I developed the formal guarantees of the core security mechanisms, the first of which was even mechanically checked in Coq.

My motivation for building secure systems is simple: security problems are everywhere. Hundreds of millions of users have had their private information (passwords, health records, credit card numbers, etc.) compromised in 2014 alone [2]. This is because building secure software is an error-prone task; existing programming models make it easy to write insecure code and notoriously difficult to produce secure code. Unfortunately, attempts to address this by creating a culture of good security practices and fixing vulnerabilities post-hoc are not working. Even seasoned Linux kernel developers have committed changes—sometimes as small as a single line—that have led to vulnerabilities [21]. How then can we expect the average programmer to build secure systems?

My research tackles this challenge by exploring new system design points that change programmer behavior in favor of producing secure code. The most effective way to change programmer behavior is to change programming languages and APIs. From the release of Java, to Ruby on Rails, to iOS and Android, history has shown that programming languages and APIs can have a profound impact on programmer behavior. Hence, by designing languages and APIs with security in mind, we can make it easier for developers to write secure code and restrict the damage that results from inevitable mistakes. For these abstractions and mechanisms to be usable by average developers, they must *adoptable*. Equally important, they should be *principled*, so as to allow one to apply formal reasoning to rule out large classes of design error.

My general approach to building secure systems is as follows. First, I find it useful to design or model a system from a clean slate. This helps to identify and understand the core problem, free from the significant real-world constraints of usability and backwards-compatibility. Importantly, it also allows for the design to be based on fundamental principles, such as information flow control [13], from the start. Then, I implement a prototype and develop the formal semantics and guarantees for the security mechanism. These steps alone are not unique. Indeed, they are common when one designs research programming languages, but by virtue of having eschewed compatibility, such principled designs are hard to apply to real systems.

By contrast, the path to adoption is much clearer when security solutions are developed as incremental improvements to existing systems. While high-impact, such solutions tend not to be principled. I attempt to get the best of both worlds through an iterative design process that refines a clean-slate design according to real-world usage and concerns.

To this end, I build real applications on top of my research systems and, when possible, get inexperienced developers to do so. I have found that less experienced developers can provide invaluable insights into the obstacles that a system such as Hails faces for adoption by average developers.

Of equal importance is to understand how developers structure applications and think about security for insecure "legacy" systems, and what changes can be made to such systems in order to incorporate the abstractions and security mechanisms that have arisen from clean-slate research. To this end, I often find it useful to reach out to individual developers and get involved in developer communities. For example, one of the factors that has made COWL attractive for standardization is its re-purposing of existing browser concepts and mechanisms to provide strong security guarantees. The idea for this came out of trying to explain information flow control to application and browser developers while I was working at Mozilla.

My thesis research applies this iterative design approach to address web application security. Web applications face security issues on two fronts, server-side and browser-side, and vulnerabilities on two sides have a multiplicative effect. Worse still, the evolution of the Web has prioritized functionality over security and led to the development of brittle and ad-hoc security solutions on both sides. Hails and COWL, together, provide end-to-end security against the privacy leaks that plague today's applications, without trading off functionality. Indeed, the strong security opens up the possibility of deploying applications that, because of security concerns, were not previously practical. These systems and their underlying security mechanisms are described below.

**Hails.** Hails [6] is a Haskell web framework designed to make it more difficult to write vulnerable code. Today, even high-profile web sites are vulnerable to application-level attacks—e.g., Github had a vulnerability that allowed a user to set the authentication keys for any project on the site [10], while Snapchat was vulnerable to an attack that allowed any user to extract the username and phone number of any other user [4]. Such vulnerabilities arise so frequently because web sites specify and enforce security policy by strewing checks throughout the application code. Overlooking even a single check can lead to vulnerabilities.

Yet, for important and sensitive data, developers typically have a more declarative, high-level security policy in mind—e.g., "a user's credit card number should not be sent to the network," or "a user's email address should only be seen by her friends." Hails allows developers to specify such data access policies alongside data schemas, where developers already specify the format of data and how it should be stored.

The framework then enforces these policies system-wide, in a mandatory fashion, using language-level information flow control. This means that policies follow data (as it leaves the database) through all software components and, even when buggy or malicious, these components cannot leak data. The Hails underlying security mechanism, LIO [15–17], ensures that all code abides by the policy.

In Hails, application logic code does not need to be intertwined with security checks. This code solely needs to implement the site functionality; the framework enforces all the security policies. Indeed, the application logic code can even be written by untrusted third-party developers.

Hails has been used to build several secure web sites, by developers with a wide-range of expertise, from a novice high school student to expert web developers. The experiences from building such applications have, in turn, been used to fine-tune the framework API and underlying security mechanism. For example, I developed a declarative policy specification language for Hails to address difficulties with specifying policies imperatively. More recently, I ported Hails to JavaScript and co-founded a company called GitStar that is developing a platform for deploying both Haskell and JavaScript web sites that wish to offload security concerns to Hails. My goal here is to influence the broader web developer community and learn how the Hails security model can scale to applications with larger code bases and unforeseen needs.

**LIO.** Hails relies on language-level information flow control (IFC) to enforce application-specific policies at runtime. A significant challenge lies in designing an IFC system that is both principled and adoptable. Many IFC programming languages are principled but lack features crucial to building real systems (e.g., exceptions, concurrency, policy inspection, and recovery from IFC-monitor failures). By contrast, many IFC operating systems are practical but lack formal semantics and cannot be easily adopted by web developers (e.g., they require a new OS and lack support for fine-grained policies common to web applications).

In an attempt to get the best from both worlds, I developed LIO [15–17], a dynamic language-level IFC system that shares many abstractions with OS-level IFC systems. By exploring a new design point in language-level IFC, LIO supports many modern language features to which developers have grown accustomed, including exceptions and concurrency. Most dynamic IFC programming languages lack such features due to the covert channels that can arise as a result of complex program control flow, but LIO eliminates these covert channels by construction [7, 16, 17].

I have formalized LIO using small-step semantics and proved that any program written in LIO cannot leak data by abusing language-level features; the proof for the sequential LIO system has been mechanically checked in Coq. The concurrent LIO design is the first dynamic IFC language to provide this result without hampering flexibility (e.g., by disallowing branching on secrets). Furthermore, to reduce the gap between language semantics and implementation, which can sometimes allow for real leaks that the semantic model does not capture, I extended LIO's semantics (and, in turn, implementation) to account for subtle implementation details, such as caches [18].

LIO has been implemented as a Haskell library and, more recently, the ideas have been applied to JavaScript in Node.js [7]. The library approach has allowed me to evaluate different design points with rapid feedback. Indeed, LIO has been rewritten several times to address challenges and limitations that arose when building Hails. In addition to Hails, however, LIO has been used to build several secure applications and systems. Finally, LIO has been part of the curricula at Stanford (Functional Systems in Haskell) and UPenn (Advanced Programming), and has served as a research platform at Chalmers, Harvard, MIT Lincoln Labs, and University of Maryland.

**COWL.** While Hails/LIO provides strong security on the server side, COWL [19] enforces security in the browser. Large parts of modern web sites are typically implemented in JavaScript that runs in the browser. These browser-side *applications* routinely incorporate code from third-parties—e.g., jQuery is used by over 57% of the top 10,000 sites [3]. Yet, in the status-quo browser, these libraries run with the privilege of the page and must be trusted to not leak the user's sensitive information. Unfortunately, even trustworthy libraries put the user's privacy at risk—e.g., jQuery's web servers were recently compromised [9] and could have been used to serve a malicious library.

COWL is a JavaScript confinement system that extends the browser security model with information flow control, while retaining backwards compatibility. Much like Hails/LIO, this allows developers to associate policy with sensitive data, such as passwords. Within the confines of the browser, COWL then enforces these policies by prohibiting code, even a malicious jQuery, from arbitrarily leaking data.

As with Hails and LIO, finding abstractions that developers can use to build applications more easily was crucial. To this end, COWL adopted the Hails/LIO abstractions to the browser. Importantly, it did so by retrofitting existing browser concepts and constructs.

For example, in the existing model, developers already express policy in terms of *origins* (the address of a web server) and compartmentalize applications using browsing contexts (e.g., iframes). COWL leverages origins to provide a simple policy model [14] that developers can use to protect sensitive data. It then enforces policies at context boundaries—e.g., when an iframe communicates with the network or another page. This has the added benefit of allowing one to implement IFC by repurposing existing security mechanisms, such as content security policy (CSP).

COWL has been implemented in both Firefox and Chromium. Several secure applications have been written using COWL, including a password manager, an encrypted document editor, and a third-party personal finance mash-up (browser-side `mint.com`). These efforts revealed multiple design and implementation bugs in HTML5 and CSP, for which I proposed new security directives that will appear in the next version of the W3C CSP specification.

The W3C Web Application Security Working Group [20] is currently in the process of standardizing COWL and I am the editor of the specification.

**Future work.** There are several directions in which I wish to continue my work on secure systems:

*Least privileged systems:* Iterating on the LIO and COWL designs, I recently started developing a language-level security architecture for Node.js called ESpectro. The goal of ESpectro is to allow developers to build application that are *least privileged*, i.e., applications where code operates using the least set of privileges necessary to complete its function. To this end, ESpectro provides developers with a way to execute untrusted JavaScript in light-weight isolated compartments, similar to COWL's browsing contexts. Within a compartment, code only has access to virtualized libraries exposed by the compartment's parent.

This simple abstraction allows developers to expose different APIs and security mechanisms, including LIO-style IFC and fine-grained discretionary access control, as libraries.

While this is still ongoing work, ESpectro is already being used by GitStar to provide a Hails-like framework for server-side JavaScript. I am currently investigating how this architecture can generalize to other language runtimes, e.g., PHP and Python.

The Breach browser [1] is integrating ESpectro to ensure that different JavaScript modules, which implement core parts of the browser (e.g., the address bar), run with least privilege. I am generally interested in exploring security mechanisms and policy languages that can allow such developers to build secure applications more easily.

Finally, I am interested in exploring a clean slate approach to building secure low-level systems and applications. Building secure systems applications, such as the exemplary privilege-separated secure shell (SSH) [12] and OKWS web server [8], is notoriously difficult. A preliminary thought is to design a language that allows programmers to describe system components (HTTP parser, logger, etc.), typed interfaces between the components, and high-level security policies. Given such a description, a compiler can then generate the different isolated components, interfaces, and mechanisms. (Such a compiler can potentially leverage existing program partitioning work, such as [11, 23], to ensure that the partitioning is efficient and secure.) The programmer would in turn only need to "fill in" the component functionality (e.g., algorithm for parsing HTTP). I am interested in developing this in the context of the type- and memory-safe systems language Rust, especially because the Rust team's interest in building secure systems could serve as a way to iterate on design.

*Policy synthesis.* Incorrect policies can break functionality or, worse, lead to leaks. I am interested in developing tools and design patterns that can help developers specify correct policies more easily. One promising approach is to use program synthesis as a way to generate policies from user-supplied examples of failed and successful access patterns. This can serve the dual role of ensuring that an already specified policy leads to expected behavior.

*Browser security.* I am generally interested in building secure browser engines. In the current monolithic browser design, new features and APIs are added as trusted code into the core browser engine. In addition to increasing the attack surface, reasoning about security when adding new features is very difficult—e.g., should iframes access geolocation data? In [22], I argued for the adoption of a unified underlying security mechanism—namely, IFC—that would address many of these concerns. In addition to exploring this idea further, I am currently investigating new browser-extension architectures, using the Breach browser as a prototype vehicle, to address user privacy in the presence of vulnerable and malicious extensions. More generally, I am interested in exploring new browser designs: are there a few core abstractions that we can use to implement most modern browser features as untrusted modules?

*Security foundations.* Finally, building on [5], I am currently developing encodings that show equivalences between coarse-grained OS-like IFC, LIO-style IFC, and fine-grained IFC programming languages. In this research area, I am more broadly interested in exploring the relationship between different security mechanisms (e.g., IFC, capabilities, discretionary access control, and role based access control), finding semantic definitions and models for least privileged systems, and formally reasoning about end-to-end security in the presence of declassification, i.e., explicit and safe "leaks."

The rise of new application domains, platforms, and community building tools, such Stack Exchange, has accelerated the speed and willingness of developers to adopt new languages and features. By capitalizing on these trends, language-based security can have real impact on systems. Indeed, there are many longstanding open problems in systems security that can be addressed using programming language methods. I want to continue to tackle these problems in an academic research setting by building practical systems that can leverage ideas from programming languages in novel ways.

## References

[1] Breach - a new modular browser. `http://breach.cc/`, Dec. 2014.

[2] Have i been pwned? check if your email has been compromised in a data breach. `https://haveibeenpwned.com/`, Dec. 2014.

[3] jQuery Usage Statistics: Websites using jQuery. `http:`

//trends.builtwith.com/javascript/jQuery, Sept. 2014.

[4] Troy hunt: Searching the snapchat data breach with "have i been pwned?". http://www.troyhunt.com/2014/01/searching-snapchat-data-breach-with.html, Jan. 2014.

[5] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. In *Computer Security Foundations Symposium (CSF)*. IEEE, July 2014.

[6] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Oct. 2012.

[7] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control, Apr. 2015. Accepted.

[8] M. N. Krohn. Building secure high-performance web services with okws. In *USENIX Annual Technical Conference (ATC), General Track*, June 2004.

[9] M. Kumar. jquery official website compromised to serve malware. http://thehackernews.com/2014/09/jquery-official-website-compromised-to.html, Sept. 2014.

[10] L. Latif. Github suffers a Ruby on Rails public key vulnerability, Mar. 2012. http://www.theinquirer.net/inquirer/news/2157093/github-suffers-ruby-rails-public-key-vulnerability.

[11] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Symposium on Operating Systems Principles (SOSP)*. ACM, Oct. 2009.

[12] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, Aug. 2003.

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

[14] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*. Springer, Oct. 2011.

[15] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, Sept. 2011.

[16] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, Sept. 2012.

[17] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming (JFP)*, 2012. Accepted/under revision.

[18] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, Sept. 2013.

[19] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining JavaScript with COWL. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Oct. 2014.

[20] W3C. Web application security working group charter. https://w3c.github.io/webappsec/admin/webappsec-charter-2015.html, Dec. 2014.

[21] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with kint. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Oct. 2012.

[22] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX, May 2013.

[23] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3), 2002.