Stephen Chong
Research Statement

August 2015

My research makes it easier to build computer systems that handle sensitive information correctly. Society increasingly relies on computer systems that handle sensitive information: government, businesses, military, and private individuals all use computer systems that manipulate confidential and/or untrusted data. For example: health management systems with confidential patient records, web applications with data supplied by potentially malicious users, and mobile devices with private user data. Sensitive information—both confidential and untrusted—must be treated carefully: confidential information must not be inappropriately released, and the use of untrusted information must not corrupt trusted computation.

However, "handling sensitive information correctly" depends in large part on the specifics of the application. For example, the restrictions on who may learn about a patient's medical records are vastly different from who may learn about the financial records of a privately-held company. Thus, key challenges in this area are to develop general techniques and tools to: (1) *define application-specific security guarantees*; (2) *express and understand the security requirements of applications;* and (3) *enforce application-specific security requirements*.

My research goal is to develop tools and techniques that allow developers of computer systems **to express application-specific information security requirements**, and **to efficiently and correctly enforce those requirements** in the computer systems they build. Moreover, the tools and techniques should ideally **clearly and formally characterize the security guarantees** they achieve, including the assumptions on which the security guarantees rely.

My research area is *language-based security*: the use of programming language concepts and techniques for security. Language-based approaches to security provide several benefits over other approaches [36]. First, a rich tradition of formal methods in programming language research enables clear and unambiguous definitions of computer security and the possibility of rigorous formal proofs that enforcement mechanisms correctly enforce security. Formal methods can prove that a mechanism provides security guarantees against entire classes of attacks [35], which is a strong guarantee against currently unknown attacks that adversaries might mount in the future. Second, since developers use programming languages to build computer systems, language-based approaches to security often lead to practical tools and techniques. Third, language-based security is well-suited to specification and enforcement of application-specific security because application-meaningful abstractions are typically present at the level of programming languages. By contrast, lower levels of abstraction such as operating systems or networks are unable to easily identify application-meaningful concepts. For example, consider trying to prevent credit card numbers from leaving a computer system. It is relatively easy to identify the program variables that contain credit card numbers and restrict their use. However, it is difficult for a network firewall to identify such data in network packets, since the network knows little about the information contained in the packets.

My work takes advantage of these benefits of language-based security. In pursuing my research goal, I strike a balance between well-principled formal security guarantees and practical, scalable tools and techniques that provide security for real programs.

In the remainder of this document, I summarize selected recent research contributions, categorized by whether the research primarily: provides foundational contributions to defining information security; supports expressing or understanding information security policies; or enforces security guarantees. I also briefly describe ongoing and future research.

1 Defining Security

To ensure that the systems we build are secure, we first need to understand what it means for a system to handle sensitive information securely. A formal semantic condition can provide a clear, unambiguous, and (ideally) intuitive notion of security. Moreover, formal definitions enable us to determine whether an enforcement mechanism correctly enforces security.

Noninterference-based semantic security conditions [19, 33] were among the first strong information security guarantees to be enforced with language-based techniques [17, 40]. Intuitively, noninterference requires that sensitive inputs to a system cannot influence non-sensitive outputs. That is, outputs to public channels should not be influenced by confidential data and the results of trusted computation should not be influenced by untrusted data. However, in practice, noninterference is insufficiently expressive to capture the security requirements of real applications. For example, many applications *declassify*, or *release*, confidential information as part of their intended functionality, which violates noninterference. Also, often applications must *erase* or *delete* sensitive information at certain times. These security requirements cannot be expressed just by simple noninterference conditions.

This area of my research examines strong semantic security conditions that go beyond noninterference to address the security requirements of realistic applications.

Dynamic security policies In many computer systems, the security policy for sensitive information changes over time. For example, as employees within a company join, leave, or change roles, the information they are permitted to learn changes.

With Aslan Askarov (then a post-doctoral researcher in my group, now at Aarhus University), I investigated how to define security in the presence of dynamically changing security policies. That is, what does it mean for a system to be secure when the policy to enforce may change during execution? We provide a simple and intuitive knowledge-based security condition: an observer should learn information only in accordance with the currently enforced security policy. However, this simple and well-motivated formal security condition has a counterintuitive consequence: security against the most powerful possible attacker does not imply security against all attackers. This is a foundational result: if securing confidentiality information ultimately relies on restricting what attackers learn, then we must restrict how attackers' knowledge changes, which depends on the abilities of the attacker to observe and remember. This work was published at the 2012 Computer Security Foundations Symposium (CSF) [2].

Required information release Many computer systems violate noninterference due to the intentional release of confidential information. Much work has considered weakening noninterference to permit some flow of confidential inputs to public outputs. However, many systems have more than just *permission* to release information; they have an *obligation* to release information. In general, transparency of organizations and processes requires the release of sensitive information. For example, at the end of a sealed bid auction, the winning bid (and, depending on the auction, the winner's identity) is required to be released.

I introduced the semantic security condition of *required information release* and demonstrated enforcement techniques for it. The definition of required information release is based on *algorithmic knowledge* [20]. Intuitively, a system satisfies required information release if an observer of the output has a specific (and, ideally, simple) algorithm to learn the released information. Thus, a required information release policy specifies both *what* information is to be released and *how* it is to be learned by an observer. These knowledge algorithms are analogous to instructions in user manuals or to self-explanatory output. Required information release interacts elegantly with permitted information release: permitted information release can be regarded as providing an upper bound on the information an observer may learn; required information release provides a lower bound, specifying information that an observer must be able to learn. This work was published at the 2010 Computer Security Foundations Symposium (CSF) [9] and was subsequently invited to appear in the Journal of Computer Security [10].

2 Expressing and Understanding Security

Pidgin With graduate students Andrew Johnson, Lucas Waye, and Scott Moore, we developed Pidgin [23], a program analysis tool that enables the specification (and enforcement) of precise application-specific information security guarantees. Pidgin uses *program dependence graphs* (PDGs) to precisely describe how information flows within a system. Properties of paths in a PDG correspond to global information security guarantees. Using a custom graph query language, Pidgin users can query PDGs to explore and specify the information security guarantees their programs provide. The query language is flexible and open-ended, allowing users to easily describe application-specific security guarantees.

Pidgin is a practical tool. We invested significant engineering effort to ensure it scales well for real programs. The query language is expressive, supporting a large class of precise, application-specific security guarantees. We have used Pidgin to: (1) explore information security guarantees in legacy programs; (2) develop and modify security policies simultaneously with application development; and (3) develop security policies based on known vulnerabilities, suitable for regression tests.

By enabling open-ended exploration of an application's security guarantees, Pidgin takes a markedly different approach than previous techniques (such as security-type systems [40, 33]) and other tools that use PDGs for security (e.g., [21]). Moreover, Pidgin separates security policies from code, and so does not prevent development or testing of the application.

Work on Pidgin was inspired by earlier work with post-doctoral researcher Jeffrey Vaughan (now at Google) on inference of *declassification policies* that describe what confidential information a program might reveal [38]. This work appeared at 2011 IEEE Symposium on Security and Privacy.

Security Guarantees from Software Architecture With collaborator Ron van der Meyden (University of New South Wales), I have shown that application-specific information-security guarantees can be proved from an abstract architectural description of an application [13]. From a description of permitted communication between components (with some additional local restrictions on trusted components), it is posible to specify and prove strong information security guarantees, expressed as formulas in epistemic logic. The insight of this work is that understanding the security guarantees of a system at the architectural level simplifies enforcement: any implementation that conforms to the architecture will enjoy the security guarantees, and so enforcement of system-wide security is reduced to architectural conformance and enforcement of local restrictions on trusted components.

3 Enforcing Security

Ideally, enforcement mechanisms should be sound, usable, and precise. Language-based enforcements can enjoy all of these properties: formal language models enable soundness, i.e., provably correct enforcement; programming language features can present programmer-friendly security abstractions; and the availability of application-meaningful abstractions can enable precise enforcement of application-specific security requirements. We have considered practical enforcement of strong information security using a variety of techniques (including capabilities, hybrid monitors, cryptographic mechanisms, and PDGs) in a variety of settings (including concurrent programs and web applications).

Capabilities A *capability* is a reified representation of the authority to perform potentially dangerous actions in a computer system. Capabilities have been used in many systems to control the authority of users and computer processes.

We have incorporated capabilities into programming languages to provide simple and intuitive mechanisms to restrict the authority of programs, thereby helping prevent applications from violating security requirements.

With postdoctoral researcher Christos Dimoulas, and graduate students Scott Moore and Dan King, we have developed Shill, a secure shell scripting language. Shill programs combine language-level capabilities with software contracts. The contracts are essentially declarative security policies, describing the capabilities that a program requires, and how the program might use the capabilities. Through language design and kernel-level sandboxing, Shill limits the effects of program execution to only actions permitted by capabilities the program possesses. The combination of capabilities with software contracts enables compositional reasoning about the potential effects of programs.

We have implemented a prototype of Shill for FreeBSD.[1] Shill is a practical and useful system security tool, providing fine-grained security guarantees and protecting systems against untrustworthy programs. This work was published at the 2014 USENIX Symposium on Operating Systems Design and Implementation (OSDI) [29]. Despite the recency of this work, other researchers have been inspired by it and enforced similar restrictions on capabilities in other language settings [39].

Effective use of capabilities as a security mechanism requires the ability to reason about and restrict the propagation of capabilities within a system. In further work, we have used declarative policies on capabilities to control their propagation [18]. We allow the developer to declare which software components are permitted to possess a given capability and automatically enforce this security policy by an access control mechanism. Moreover, the developer can declare which components are permitted to *influence* the use of a given capability, leading to a non-interference-like security condition that can be achieved using standard information-flow control mechanisms. This work was published at the 2014 Computer Security Foundations Symposium (CSF).

Hybrid information-flow control techniques By controlling the flow of information in computer systems, strong, precise, application-specific information security (such as noninterference) can be enforced [33]. *Hybrid information-flow security monitors* (e.g., [37, 24, 5, 32]) combine static and dynamic techniques for language-based information-flow control and enjoy benefits of both static and dynamic approaches.

We discovered theoretical limits on precision of hybrid information-flow control, by extending a hybrid monitor [32] to handle dynamically allocated memory. Graduate student Scott Moore and I identified sufficient conditions for soundness on memory abstractions that the monitor uses to track information flow through dynamically allocated memory. Interestingly, we found that certain memory abstractions are unsound due to being too precise. This led to the discovery of an

[1]Available at `http://shill.seas.harvard.edu/`.

unintentional source of unsoundness in an existing hybrid information-flow monitor. This work appeared at the 2011 Computer Security Foundations Symposium (CSF) [27].

We additionally developed hybrid information-flow monitors to precisely enforce termination-sensitive security (i.e., preventing the revelation of confidential information by the termination behavior of programs). With Askarov and Moore, I show that contrary to common belief, it is possible to enforce termination-sensitive security without overly restrictive enforcement mechanisms: a case study of a security-typed program showed that the termination behavior of all (intraprocedural) loops depends only on non-confidential information and are amenable to existing termination analyses. This work was published at the 2012 ACM Conference on Computer and Communications Security (CCS) [28].

With collaborator Heiko Mantel (TU Darmstadt) and Askarov, I further extended the state of the art of hybrid information-flow monitoring by provably and efficiently enforcing termination-sensitive security in concurrent programs [3]. In this work, each thread has its own local monitor that tracks and restricts information flow within the thread, and a single global monitor ensures appropriate co-ordination between threads. Importantly, the local monitors communicate with the global monitor only when threads synchronize. This innovation ensures that the monitor does not unnecessarily restrict concurrency. This work was published at the 2015 Computer Security Foundations Symposium (CSF).

Concurrency Due to increasing hardware parallelism, concurrent systems that handle sensitive information are common, and we need practical mechanisms to enforce strong security in such systems. Since both information security and concurrency are connected to notions of dependency [1], there is potential for synergy between language mechanisms for concurrency and enforcement mechanisms for information security in concurrent programs.

With undergraduate Stefan Muller, I exploited such synergy in the X10 programming language, published at OOPSLA 2012 [30]. In X10, a *place* is a computational unit that contains computation and data (such as a core of a machine). Multiple threads may execute concurrently within a place. Threads at the same place share memory, and a thread may access data only at the place where it is located. Communication between places is by message passing. We extend X10 with coarse-grained information security mechanisms: we associate each place with a security level, and use a (completely static) security analysis to ensure that each place stores only data appropriate for that security level. Thus, all computation within a place is on data at the same security level. Interaction between places may influence the scheduling of threads at a place, leading to potential covert information channels; our security analysis tracks and controls these covert channels using an existing may-happen-in-parallel analysis for X10 [25]. The key innovation here is to leverage recently developed language mechanisms for concurrency to provide simple, practical, and useful abstractions for strong information security in concurrent programs.

Additional work using hybrid information-flow security monitors to efficiently enforce security in concurrent programs [3] was described above.

Web Application Security It is difficult to develop secure web applications using existing languages and frameworks. The dynamic and distributed nature of web applications is difficult to reason about, leading to numerous and costly vulnerabilities.

One challenge is to correctly sanitize untrusted data. Failure to do so can lead to injection vulnerabilities, and it is notoriously difficult for developers of large web applications to get it right [34]. With collaborator Benjamin Livshits (Microsoft Research), I proposed a novel technique for the automated placement of sanitizers in web applications [26]. The developer simply states what sanitization needs to be applied to data as it moves from sources to sinks, and our mechanism automatically modifies the code to place sanitizers on appropriate code paths, ensuring correct sanitization. Placement is static whenever possible, but switches to using run-time taint tracking techniques when necessary for correctness. This work was published in the 2013 Symposium on Principles of Programming Languages (POPL).

Information Erasure Information erasure is a formal security requirement that stipulates when sensitive data must be removed from computer systems. I introduced this security requirement in earlier work [11, 12], and other researchers have refined it and developed enforcement mechanisms (e.g., [22, 31]). In a system that correctly enforces erasure requirements, an attacker who observes the system after sensitive data is erased cannot deduce anything about the data.

Practical obstacles to enforcing information erasure include: (1) correctly determining which data requires erasure; and (2) reliably deleting potentially large volumes of data, despite untrustworthy storage services. The first obstacle can be overcome using language-based information-flow control mechanisms. With collaborator Askarov, graduate student Moore, and postdoctoral researcher Dimoulas, we formalized the use of cryptographic mechanisms to address the second obstacle: sensitive data is encrypted before storage, and upon erasure, only a relatively small set of decryption keys needs to be deleted. Although this cryptographic technique is used by a number of systems, we combine it with language-based information security mechanisms to support the correct determination of data that needs erasure. This novel combination also improves the efficiency of existing language-based mechanisms for information erasure. This work was published at the 2015 Computer Security Foundations Symposium (CSF) [4].

4 Additional Research

Although primarily focused on language-based information security, I am also active in other areas of programming language research. I describe two such projects in this section.

Functional Reactive Programming The Elm programming language[2] is a functional reactive programming language (FRP) for graphical user interfaces. Developed under my supervision by undergraduate Evan Czaplicki for his senior thesis, Elm provides elegant programming abstractions for developing web-based clients. A paper published at the 2013 conference on Programming Language Design and Implementation (PLDI) presents a core calculus for Elm, including a novel language feature for *asynchronous FRP* that enables the efficient concurrent execution of FRP programs [16]. Elm has been used in programming language courses at the University of Chicago and KU Leuven, is used by at least 5 companies, and is the language used by McMaster University's software outreach program. Moreover, Elm has over 100 community-contributed packages.

Provenance Provenance is metadata about the origin, context, or history of data. Provenance can enable many use cases, ranging from replication of computational experiments to *post facto* auditing and authentication. However, the collection, maintenance, and use of provenance is challenging. I have defined semantic security conditions for provenance [8], embedding provenance in datasets [14, 15] (with collaborators Jeffrey Vaughan, Google, and Christian Skalka, University of Vermont), and co-authored a provenance research vision paper [6].

[2]`http://elm-lang.org/`

5 Ongoing and Future Work

My group and collaborators have a number of active projects, many focused on providing strong application-specific information security guarantees. Projects include formalization of privacy legislation using logic programming languages, provably correct auditing, software contracts for distributed applications, and the use of new hardware architecture mechanisms to enforce information erasure. Two additional projects are described below.

Enforcing language semantics in distributed applications Proof-carrying Data (PCD) [7] is a recent cryptographic mechanism for verified computation. With collaborators Eran Tromer (Tel Aviv University) and Jeffrey Vaughan (Google), I am incorporating PCD into an object-oriented language in order to enforce language semantics in distributed applications. When a party receives an object from an untrusted sender, if the object is accompanied by an appropriate PCD proof then the recipient knows that the object was computed in accordance with language semantics (and thus, for example, may infer that hard-to-check object invariants hold). The recipient does not know what computation produced the object, allowing the sender to perform the computation using private inputs and proprietary algorithms. This allows programmers to soundly reason about program behavior, despite values received from untrusted parties, without needing to be aware of the underlying cryptographic techniques.

Authority control With graduate student Moore and postdoctoral researcher Dimoulas, I continue to investigate language-based mechanisms to control authority, furthering the work started with Shill and our capability-focused research. Our current research exploits the insight that dynamically and lexically scoped authority (analogous to dynamic and lexical variable scope) is a simple primitive mechanism that is sufficient to implement a wide range of existing authority control mechanisms, such as discretionary access control, stack inspection, and coarse-grained information-flow control. By providing a common framework that allows the creation of *authority closures* (i.e., computations with lexically scoped authority), we can implement several authority-control mechanisms, and provide guarantees when these mechanisms are combined in various ways.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322. IEEE Press, June 2012.

[3] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*. IEEE Press, July 2015.

[4] Aslan Askarov, Scott Moore, Christos Dimoulas, and Stephen Chong. Cryptographic enforcement of language-based erasure. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*. IEEE Press, July 2015.

[5] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforce-

ment in a Java virtual machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[6] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: A future history. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications: Onward! Session*, pages 957–964, October 2009.

[7] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Proceedings of the 2010 Conference on Innovations in Computer Science*, pages 310–331, 2010.

[8] Stephen Chong. Towards semantics for provenance security. In *Proceedings of the 1st Workshop on the Theory and Practice of Provenance*, February 2009.

[9] Stephen Chong. Required information release. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 215–227. IEEE Press, July 2010.

[10] Stephen Chong. Required information release. *Journal of Computer Security*, 20(6):637–676, 2012.

[11] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 241–254. IEEE Press, June 2005.

[12] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111. IEEE Press, June 2008.

[13] Stephen Chong and Ron van der Meyden. Using architecture to reason about information security. In *Proceedings of the 6th Layered Assurance Workshop*, pages 1–11, 2012.

[14] Stephen Chong, Christian Skalka, and Jeffrey A. Vaughan. Self-identifying sensor data. In *Proceedings of the Ninth International Conference on Information Processing in Sensor Networks*, pages 82–93, April 2010.

[15] Stephen Chong, Christian Skalka, and Jeffrey A. Vaughan. Self-identifying data for fair use. *Journal of Data and Information Quality*, 5(3), December 2014.

[16] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422. ACM Press, June 2013.

[17] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[18] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium*. IEEE Press, June 2014.

[19] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

[20] Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Algorithmic knowledge. In *Proceedings of the 5th Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 255–266, March 1994.

[21] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009.

[22] Sebastian Hunt and David Sands. Just forget it—the semantics and enforcement of information erasure. In *Proceedings of the 17th European Symposium on Programming*, pages 239–253. Springer, 2008.

[23] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pages 291–302. ACM Press, June 2015.

[24] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. *Proceedings of the 11th Annual Asian Computing Science Conference*, pages 75–89, 2006.

[25] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 25–36, New York, NY, USA, January 2010. ACM.

[26] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 385–398. ACM Press, January 2013.

[27] Scott Moore and Stephen Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium*, pages 146–160. IEEE Press, June 2011.

[28] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 881–893. ACM Press, October 2012.

[29] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, October 2014.

[30] Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, pages 57–74. ACM Press, October 2012.

[31] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2), 2013.

[32] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

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

[34] Prateek Saxena, David Molnar, and Benjamin Livshits. ScriptGard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the Conference on Computer and Communications Security*, October 2011.

[35] Fred B. Schneider. Blueprint for a science of cybersecurity. *The Next Wave*, 19(2):47–57, 2012.

[36] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101, London, UK, 2000. Springer-Verlag.

[37] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 203–217. IEEE Computer Society, 2007.

[38] Jeffrey A. Vaughan and Stephen Chong. Inference of expressive declassification policies. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 180–195. IEEE Press, May 2011.

[39] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM International Conference on Functional Programming*, 2015.

[40] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.