

Programming Languages and Program Analysis for Security

A Three-Year Retrospective

Marco Pistoia

IBM T. J. Watson Research Center
pistoia@us.ibm.com

Úlfar Erlingsson

Reykjavík University
ulfar@ru.is

Abstract

Software security has been traditionally enforced at the level of operating systems. However, operating systems have become increasingly large and complex, and it is very difficult—if not impossible—to enforce software security solely through them. Moreover, operating-system security allows dealing primarily with access-control policies on resources such as files and network connections. However, attacks may happen at both lower and higher levels of abstraction, and may target the internal behavior of applications, such as today’s Web-based applications. Therefore, defenses must offer protection at the level of applications. *Language-based security* is the area of research that studies how to enforce application-level security using programming-language and program-analysis techniques. This area of research has become very active with the advent of Web applications. In 2006, the ACM SIGPLAN has introduced a new yearly forum entirely dedicated to the discussion of language-based-security research: Programming Languages and Analysis for Security (PLAS). This paper is a three-year survey of PLAS papers that discusses the progress made in the area of language-based security.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Program Analysis, Programming Languages, Application Security

Keywords Security, Program Analysis, Programming Languages, Language-Based Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2008 ACM [to be supplied]...\$5.00

1. Introduction

The security of a software system is almost always retrofitted to an afterthought. When security problems arise in software programs, understanding and correcting those problems can be very challenging. On the one hand, the program-analysis and programming-languages research community has designed new languages with different support for intrinsic guarantees, and created numerous static and dynamic analysis tools for software performance optimization and software bug detection. On the other hand, the security and privacy research community has been looking for solutions to automatically detect security problems, such as access-control inconsistencies, information-flow vulnerabilities and privacy violations. The ACM SIGPLAN yearly workshop on Programming Languages and Analysis for Security (PLAS) was introduced in 2006 to allow these two communities to meet and exchange ideas on how to leverage the design and implementation of programming languages and program-analysis techniques to enhance the security of software systems. This paper is a three-year retrospective on PLAS. It summarizes a set of representative PLAS papers and discusses the enhancements that have been made through PLAS in the following areas of language-based security:

- Secure language design
- Access control
- Web application security
- Secure information flow
- Trusted declassification
- Secure development

2. Secure Language Design

Designing languages with security features is a very prominent direction for language-based-security research. Important work has been done in the direction of designing brand new languages that would automatically provide security features, such as code-based [16, 37, 13, 11] and subject-based [26] authentication and authorization, information flow [36, 46], and support for cryptography [38]. Another research direction has investigated how to add missing se-

curity features to existing languages (for example, how to add support for information-flow tracking and enforcement to Java [33]).

Nielsen and Schwartzbach present a domain-specific programming language for Secure Multiparty Computation (SMC), called SMC Language (SMCL) [34]. Information is a resource of considerable economic value to individuals, public administration, and private companies. Confidentiality of information is essential, but secret values can often be obtained anyway by combining confidential information from various sources. This fundamental conflict between the benefits of confidentiality and the benefits of information sharing may be overcome using the cryptographic method of SMC, where computations are performed on secret values and results are only revealed according to specific protocols. The design of SMCL bridges the gap between high-level security requirements and low-level cryptographic operations constituting an SMC platform, thus improving the efficiency and security of SMC application development. SMCL is implemented in a prototype compiler that generates Java code exploiting a distributed cryptographic runtime.

Security-typed languages such as Java Information Flow (Jif) [33] require the programmer to label variables with information-flow security policies as part of application development. The compiler then flags errors wherever information leaks may occur. Smith and Thober work on improving the usability of information-flow type systems [45]. They present a static information-flow type-inference system for Middleweight Java (MJ) that automatically infers information-flow labels, thus avoiding the need for a multitude of program annotations. Policies need only be specified on Input/Output (I/O) channels. Their type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to properly distinguish the security policies of different I/O channels. Furthermore, they prove a non-interference property [15] for programs that interactively input and output data, and they then describe a mechanism that allows users to define top-level policies, which automatically inserts the security-policy-enforcement instructions at the proper points in the program. This provides the further benefit that whomever is defining the policy does not necessarily need intimate knowledge of the program source. The motivation of this work is similar to the approach taken by Pistoia, *et al.*, who show how to automatically extract an information-flow policy from an access-control policy and enforce them at the same time without the need for the developer or system administrator to annotate the program variables [36].

Many organizations specify information release policies to describe the terms under which sensitive information may be released to other organizations. Swamy and Hicks present a new approach for ensuring that security-critical software correctly enforces its information release policy [47]. Their approach has two parts: first, an information-release policy is

specified as a security automaton written in a new language called AIR; second, an AIR policy is enforced by translating it into an Application Programming Interface (API) for programs written in λ AIR—a core formalism for a functional programming language. λ AIR uses a novel combination of dependent, affine, and singleton types to ensure that the API is used correctly. As a consequence, it is possible to certify that programs written in λ AIR meet the requirements of the original AIR policy specification.

3. Access Control

Language-based-security techniques have been extensively applied to access-control issues, for example to enforce the Principle of Least Privilege [41] by automatically detecting an optimal access-control policy [25], or to enforce the Principle of Complete Mediation [41] by automatically inserting access-control checks before a sensitive resource is accessed, or by verifying that existing checks are properly placed [14]. When enforcing the Principle of Complete Mediation, it is also important to detect Time-Of-Check/Time-Of-Use (TOCTOU) vulnerabilities, which occur when a security check is performed for a particular resource (for example, a file) before the resource is accessed, but by the time the resource is accessed, some of the information considered by the security check (for example, the name of the file) may change in a manner that invalidates the result of the check.

Fraser, *et al.* present the first use of flow-sensitive CQUAL to verify the placement of operating system authorization checks [12]. Their analysis of MINIX 3 system servers and discovery of a non-exploitable TOCTOU bug demonstrate the effectiveness of flow-sensitive CQUAL and its advantage over earlier flow-insensitive versions [52].

Hamlen, *et al.* present Mobile [18], an extension of the .NET Common Intermediate Language that supports certified Inlined Reference Monitors (IRMs) [9]. Mobile programs have the useful property that if they are well typed with respect to a declared security policy, then they are guaranteed not to violate that security policy when executed. Thus, when an IRM is expressed in Mobile, it can be certified by a simple type-checker to eliminate the need to trust the producer of the IRM. Security policies in Mobile are declarative, can involve unbounded collections of objects allocated at run time, and can regard infinite-length histories of security events exhibited by those objects. The Mobile runtime enforces properties expressed by finite-state *security automata* [42]—one automaton for each security-relevant object—and can type-check Mobile programs in the presence of exceptions, finalizers, concurrency, and non-termination. Executing Mobile programs requires no change to existing .NET virtual machine implementations, since Mobile programs consist of normal managed CIL code with extra typing annotations stored in .NET attributes.

Hamlen and Jones introduce then an Aspect-Oriented (AO) declarative security-policy-specification language for

enforcement by IRMs [17]. The semantics of the language establishes a formal connection between AO programming and IRMs wherein policy specifications denote *AO security automata*—security automata whose edge labels are encoded as pointcut expressions. The prototype language implementation enforces these security policies by automatically rewriting Java bytecode programs so as to detect and prevent policy violations at run time.

4. Web Application Security

The security issues that can arise with Web applications are mostly in the area of integrity and confidentiality [27]. To prevent integrity violations, such as Structured Query Language injection (SQLi) and Cross-site Scripting (XSS) attacks [50, 51], a Web application should sanitize all the inputs it receives from untrusted users. Similarly, to prevent confidentiality violations, a Web application should verify that the information it releases is not private or has been appropriately declassified. In recent years, the security landscape has changed, with Web applications vulnerabilities becoming more prominent than traditional vulnerabilities such as buffer overruns. Many reports point to SQLi and XSS as being the most common attacks against Web applications to date [35]. With the advent of Web 2.0, existing security problems are further exacerbated by the presence of Ajax technology, which allows continuous network activity, and the creation and composition of content from different sources within the browser at run time, as exemplified by customizable mash-up pages.

Livshits and Erlingsson propose a simple to support, yet powerful scheme for eliminating a wide range of script injection vulnerabilities in applications built on top of popular Ajax development frameworks [28]. Unlike other client-side runtime enforcement proposals, their solution requires only minor browser modifications. This is because their approach can be viewed as a natural finer-grained extension of the same-origin policy for JavaScript, which is already supported by virtually all browsers, the only difference being that individual user interface widgets are treated as belonging to separate domains. In most cases no changes to the development process need to take place: for applications that are built on top of frameworks described above, a slight framework modification will result in appropriate changes in the generated HTML, thereby eliminating the need for manual code annotation.

Zheng and Myers present a non-intrusive encryption mechanism for protecting data confidentiality on the Web [53]. The core idea is to encrypt confidential data before sending it to untrusted sites and use keystores on the Web to manage encryption keys without intervention from users. A formal language-based information flow model is used to prove the soundness of the mechanism.

Since manual code reviews are time-consuming, error-prone and costly, the need for automated solutions for find-

ing security vulnerabilities in Web applications is evident. Jovanovic, *et al.* address the problem of vulnerable Web applications by means of static source-code analysis [24]. To this end, they present a novel, precise alias analysis targeted at the unique reference semantics commonly found in scripting languages. Moreover, the quality and quantity of the generated vulnerability reports is enhanced via a novel algorithm for fast and precise resolution of file inclusions.

5. Secure Information Flow

Type systems are a useful mechanism for efficiently checking that information flows within a programs are secure. Type systems are, however, conservative. As a consequence, they often reject safe programs as ill-typed. Accordingly, users have to check whether the rejected programs indeed have insecure flows. To remedy this problem, Unno, *et al.* propose a method for automatically finding witnesses of insecure information flow [48]. In the case of confidentiality, their method reports the exact input states that actually lead to leakage of secret information. This method is a novel combination of type-based analysis and model checking. Suspicious execution paths that may cause insecure flows of information are first found by using the result of a type-based information-flow analysis, and then a model checker is used to check whether the paths are indeed unsafe.

Adding a sound information-flow security policy to an existing program is a difficult task that may require major code inspection, program analysis, and structural changes to the code of the program. Smith and Thober show how refactoring programs into distinct components of *high* and *low* security [6] is a useful methodology to aid in the production of programs with sound information-flow policies [44]. Given a program with no information flow controls, a program slicer is used to identify code that depends on *high* security inputs. *High* security code, so identified, is then refactored into a separate component, which may be accessed by the *low* security component via public method calls. A security policy that labels input data and checks the output points can then enforce the desired end-to-end security property. Controlled information releases can occur at explicit declassification points if deemed safe. The result is a well engineered program with explicit interfaces between components of different security levels.

Hristova, *et al.* describe the design, analysis, and implementation of an efficient algorithm for information-flow analysis expressed using a type system [22]. Given a program and an environment of security classes for information accessed by the program, the algorithm checks whether the program is well typed, and there is no information of *higher* security classes flowing into places of *lower* security classes according to a lattice of security classes [6], by inferring the *highest* or *lowest* security class as appropriate for each program node. The analysis is expressed as a set of Datalog-like rules based on the typing and subtyping rules. They use

a systematic method to generate specialized algorithms and data structures directly from the Datalog-like rules. The generated implementation traverses the program multiple times and uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The time complexity of the algorithm is linear in the size of the input program times the height of the lattice of security classes, plus a small overhead for preprocessing the security classes.

In secure-information-flow analysis, the classic Denning restrictions allow a program's termination to be affected by the values of its *high* variables [7], resulting in potential information leaks. In an effort to quantify such leaks, Smith and Alpízar study a simple imperative language with random assignments. They take into account a *stripping* operation on programs and establish a fundamental relationship between the behavior of a well-typed program and of its stripped version; to prove this relationship, they introduce a new notion of fast probabilistic simulation on Markov chains. As an application, they prove that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an approximate probabilistic non-interference property, provided that their probability of non-termination is small.

Information-flow analysis can prevent programs from improperly revealing secret information, and a dynamic approach can make such analysis more practical, but there has been relatively little work verifying that such analyses are sound (account for all flows in a given execution). McCamant and Ernst present a new technique for proving the soundness of dynamic information-flow analyses for policies such as end-to-end confidentiality [30]. This forms the basis for a quantitative approach to non-interference [31]. The proof technique simulates the behavior of the analyzed program with a pair of copies of the program: one has access to the secret information, and the other is responsible for output. The two copies are connected by a limited-bandwidth communication channel, and the amount of information passed on the channel bounds the amount of information disclosed, allowing it to be quantified.

Chen and Malacaria also propose a quantitative analysis of information flow [5]. Their approach applies to a multi-threaded language and is based on a probabilistic scheduler. The analysis consists of two steps. First, multi-threaded programs are translated into single-thread looping programs with a probabilistic operator. Second, an information theoretical semantics of loops with probabilistic operators is used to derive the leakage. Using this analysis classical examples of multi-threaded programs are revisited: it is shown how the analysis is able to deal with, among other, probabilistic leakage, internally observable timing leakage and leakage originated by observing intermediate states of computation.

Malacaria and Chen then extend that previous work and explore two fundamental issues in language-based security [29]. The first issue is the need for providing a quantitative

definition of information leakage valid in several attacker models. Attackers are considered having different capabilities: the strongest attacker is able to observe the value of the *low* variables at each step during the execution of a program; the weakest attacker can only observe a single *low* value at some stage of the execution. A uniform definition of leakage is given based on information theory. This definition allows formalizing and proving some intuitive relationships between the amounts leaked by the same program in different models. The second issue is that of *channel capacity*, which in security terms amounts to answering the following two questions: (a) Given a program and an observational model, what is the maximum amount that the program can leak? (b) Which input distribution causes the maximum leakage? To answer these questions, techniques from constrained non-linear optimization (mainly Lagrange multipliers) are introduced, and are shown to provide a workable solution in all the observational models considered. In the simplest setting—under minimal constraints—channel capacity is achieved by any input distribution which induces a uniform distribution on the observables.

Focardi and Centenaro also study non-interference in the setting of multi-threaded distributed programs in which threads share local memories, and multi-threaded processes communicate over an insecure network using encryption primitives to secure messages [10]. They extend a simple imperative language with cryptographic operations, and adapt to their setting the notion of patterns proposed by Abadi and Rogaway for modeling the equivalence of cryptographic expressions [1]. Based on this notion, they naturally obtain a definition of strongly secure programs corresponding to the one proposed by Sabelfeld and Sands for programs without cryptography [39], which allow them to prove compositionality of secure programs and to adapt the type system of Sabelfeld and Sands to their setting, proving its correctness.

A static path condition is a precise necessary condition for information flow between two program points. Object-oriented languages offer constructs such as dynamic dispatch, `instanceof` and exceptions. Hammer, *et al.* present an analysis of these constructs, which leads to precise path conditions operating only on the program's variables. This yields a gain in information-flow-analysis precision, allowing leverage of automatic constraint solving [19].

A subtle way to violate non-interference is through a *covert channel*—a parasitic communication channel c_1 that draws bandwidth from another channel c_2 in order to transmit information without notifying or being authorized by the designer, owner, or operator of c_2 . Covert channels can result in unauthorized information flows when exploited by malicious software. To address this problem, Shaffer, *et al.* present a precise, formal definition for covert channels [43], which relies on control-flow-dependency tracing through program execution, and extends Dennings' and subsequent classic work in secure information flow [7, 49]. A formal

security Domain Model (DM) for conducting static analysis of programs to identify covert channel vulnerabilities is described. The DM is comprised of an Invariant Model, which defines the generic concepts of program state, information flow, and covert channel rules; and an Implementation Model, which specifies the behavior of a target program. The DM is compiled from a representation of the program, written in a domain-specific Implementation Modeling Language (IML), and a specification of the security policy written in Alloy [23]. The Alloy Analyzer tool is used to perform static analysis of the DM to automatically detect potential covert-channel vulnerabilities and security-policy violations in the target program.

Chaudhuri, *et al.* are interested in information-flow security as an integrity problem for the Windows Vista operating system. Windows Vista implements an interesting model of multi-level integrity. They observe that, in this model, trusted code must participate in any information-flow attack, since trusted code orchestrates all software execution. Thus, it is possible to eliminate such attacks by statically restricting trusted code. They formalize this model by designing a type system that can efficiently enforce data-flow integrity on Windows Vista. Type checking guarantees that objects whose contents are statically trusted never contain untrusted values, regardless of what untrusted code runs in the environment. Some of Windows Vista's run-time access checks are necessary for soundness; others are redundant and can be optimized away. The interesting contribution of this work is that it can be generalized to other systems with the property that trusted code always participates in any information-flow attack.

6. Trusted Declassification

Downgrading is the process by which information whose integrity was originally suspect is verified and *endorsed*, and information that was originally confidential is *declassified*. Clearly, downgrading should be done with extreme care since it represents a deliberate violation of information-flow security. A consistent research direction in the area of language-based security deals with trusted downgrading.

Security-typed languages promise to be a powerful tool with which provably secure software applications may be developed. Programs written in these languages enforce a strong, global policy of noninterference which ensures that *high*-security data will not be observable on *low*-security channels. Because noninterference is typically too strong a property, most programs use some form of declassification to selectively leak *high* security information, for example when performing a password check or when transmitting encrypted confidential data over an unsecure wire. Unfortunately, such a declassification is often expressed as an operation within a given program, rather than as part of a global policy, making reasoning about the security implications of a policy more difficult.

Hicks, *et al.* propose performing trusted declassification by letting the security administrator specify special declassifier functions as part of the global policy [21]. In particular, individual principals declaratively specify which declassifiers they trust so that all information flows implied by the policy can be reasoned about in absence of a particular program. This approach is formalized for a Java-like language. For this system, a modified form of non-interference, called *non-interference modulo trusted methods*, is formally proved. This approach has also been implemented as an extension to Jif [33] and used to build a secure e-mail client.

Declassification policies are a key challenge for language-based information security. Although much progress has been made, different approaches to declassification tend to address different aspects. In a recent taxonomy by Sabelfeld and Sands, these aspects are referred to as the *what*, *who*, *where*, and *when* dimensions of declassification [40]. In order to avoid information laundering, it is important to combine defenses along these different dimensions. As a step in this direction, Askarov and Sabelfeld present a combination of *what* and *where* declassification policies, and show that a minor modification of a security type system from the literature (which was designed for treating the *what* dimension) effectively enforces the combination of *what* and *where* policies [3].

Instead of focusing on the different aspects of declassifications, Banerjee, *et al.* argue that key security goals addressed by accounting for those dimensions can be expressed using assertions and an auxiliary state (such as event history) [4], building on a logic for non-interference that provides for local reasoning about the heap [2].

7. Secure Development

Writing secure code is an essential step to guarantee security in the software lifecycle [32]. Hicks, *et al.* resolve information-flow leaks during application development [20]. Since information flows can be quite subtle and difficult to detect, simple error messages tend to be insufficient for finding and resolving the source of information leaks at development time; more sophisticated development tools are needed for this task. To this end, they provide a set of principles to guide the development of such tools. Furthermore, they implement a subset of these principles in an integrated development environment (IDE) for Jif, called Jifclipse, which is built on the Eclipse extensible development platform [8]. Their tool provides a Jif programmer with additional instruments to view hidden information generated by a Jif compilation, obtain fixes for errors, and get more specific information behind an error message. Jifclipse is a first step towards making application development on a security-typed language a practical process.

8. Conclusion

The ACM SIGPLAN workshop on Programming Languages and Analysis for Security (PLAS) has, in its three years, provided an important venue for work on the intersection of programming languages and software security and privacy. As this retrospective shows, the workshop has included substantial contributions and advances on the state-of-the-art in software security. The security benefits of these contributions apply to specialized security policies, general access control, as well as information flow, integrity, and declassification. The work presented at the workshop has built upon a variety of techniques, such as model checking, pointer alias analysis, type systems and formal logic. Finally, the workshop results have touched upon the entire software lifecycle, from development to deployment, and apply to both legacy software and new code, as well as to both traditional and Web-based applications. It seems assured that the PLAS workshop will continue to provide a venue for a variety of high-quality technical contributions in this critical field of study.

Acknowledgments

The authors would like to thank Michael Hicks and Steve Zdancewicz for their invaluable contributions to PLAS. Thanks also to IBM Research and Microsoft Research for sponsoring PLAS throughout the years.

References

- [1] Martin Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *Journal of Cryptology*, 20(3):395–395, 2007.
- [2] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A Logic for Information Flow in Object-Oriented Programs. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 91–102, January 2006.
- [3] Aslan Askarov and Andrei Sabelfeld. Localized Delimited Release: Combining the What and Where Dimensions of Information Release. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 53–60, San Diego, CA, USA, June 2007.
- [4] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Towards a Logical Account of Declassification. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 61–66, San Diego, CA, USA, June 2007.
- [5] Han Chen and Pasquale Malacaria. Quantitative Analysis of Leakage for Multi-threaded Programs. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 31–40, San Diego, CA, USA, June 2007.
- [6] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [7] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.
- [8] Eclipse Project, <http://www.eclipse.org>.
- [9] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, January 2004.
- [10] Riccardo Focardi and Matteo Centenaro. Information Flow Security of Multi-threaded Distributed Programs. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 113–124, Tucson, AZ, USA, June 2008.
- [11] Cédric Fournet and Andrew D. Gordon. Stack Inspection: Theory and Variants. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):360–399, 2003.
- [12] Timothy Fraser, Nick L. Petroni Jr., and William A. Arbaugh. Applying Flow-sensitive CQUAL to Verify MINIX Authorization Check Placement. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 3–6, Ottawa, ON, Canada, 2006.
- [13] Adam Freeman and Allen Jones. *Programming .NET Security*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.
- [14] Vinod Ganapathy, Dave King, Trent Jaeger, and Somesh Jha. Mining Security-Sensitive Operations in Legacy Code Using Concept Analysis. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 458–467, Minneapolis, MN, USA, May 2007.
- [15] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, May 1982. IEEE Computer Society Press.
- [16] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, December 1997.
- [17] Kevin W. Hamlen and Micah Jones. Aspect-oriented Inlined Reference Monitors. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 11–20, Tucson, AZ, USA, June 2008.
- [18] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified In-lined Reference Monitoring on .NET. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 7–16, Ottawa, ON, Canada, 2006.
- [19] Christian Hammer, Rüdiger Schaade, and Gregor Snelting. Static Path Conditions for Java. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 57–66, Tucson, AZ, USA, June 2008.
- [20] Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: Development Tools for Security-typed Languages. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 1–10, San Diego, CA, USA, June 2007.
- [21] Boniface Hicks, Dave King, Patrick McDaniel, and Michael

- Hicks. Trusted Declassification: High-level Policy for a Security-typed Language. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 65–74, 2006.
- [22] Katia Hristova, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. Efficient Type Inference for Secure Information Flow. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 85–94, Ottawa, ON, Canada, 2006.
- [23] Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [24] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 27–36, Ottawa, ON, Canada, 2006.
- [25] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.
- [26] Charlie Lai, Li Gong, Larry Koved, Anthony J. Nadalin, and Roland Schemers. User Authentication and Authorization in the Java™ Platform. In *15th Annual Computer Security Applications Conference (ACSAC 1999)*, pages 285–290, Scottsdale, AZ, USA, December 1999. IEEE Computer Security.
- [27] Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium*, Baltimore, MD, USA, July 2005.
- [28] Benjamin Livshits and Úlfar Erlingsson. Using Web Application Construction Frameworks to Protect against Code Injection Attacks. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 95–104, San Diego, CA, USA, June 2007.
- [29] Pasquale Malacaria and Han Chen. Lagrange Multipliers and Maximum Information Leakage in Different Observational Models. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 135–146, Tucson, AZ, USA, June 2008.
- [30] Stephen McCamant and Michael D. Ernst. A Simulation-based Proof Technique for Dynamic Information Flow. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 41–46, San Diego, CA, USA, June 2007.
- [31] Stephen McCamant and Michael D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 193–205, Tucson, AZ, USA, June 2008.
- [32] Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc., New York, NY, USA, January 1999.
- [33] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*, pages 228–241, San Antonio, TX, USA, January 1999.
- [34] Janus Dam Nielsen and Michael I. Schwartzbach. A Domain-specific Programming Language for Secure Multiparty Computation. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 21–30, San Diego, CA, USA, June 2007.
- [35] Open Web Application Security Project (OWASP), <http://www.owasp.org>.
- [36] Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond Stack Inspection: A Unified Access Control and Information Flow Security Model. In *28th IEEE Symposium on Security and Privacy*, pages 149–163, Oakland, CA, USA, May 2007.
- [37] Marco Pistoia, Stephen J. Fink, Robert J. Flynn, and Eran Yahav. When Role Models Have Flaws: Static Validation of Enterprise Security Policies. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 478–488, Minneapolis, MN, USA, May 2007.
- [38] Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.
- [39] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *13th IEEE Computer Security Foundations Workshop (CSFW 2000)*, pages 200–214, Cambridge, England, UK, June 2000. IEEE Computer Society.
- [40] Andrei Sabelfeld and David Sands. Dimensions and Principles of Declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW 2005)*, pages 255–269, Aix-en-Provence, France, June 2005.
- [41] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [42] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [43] Alan B. Shaffer, Mikhail Auguston, Cynthia E. Irvine, and Timothy E. Levin. A Security Domain Model to Assess Software for Exploitable Covert Channels. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 45–56, Tucson, AZ, USA, June 2008.
- [44] Scott F. Smith and Mark Thober. Refactoring Programs to Secure Information Flows. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 75–84, Ottawa, ON, Canada, 2006.
- [45] Scott F. Smith and Mark Thober. Improving Usability of Information Flow Security in Java. In *2nd Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 11–20, San Diego, CA, USA, June 2007.
- [46] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A Language for Enforcing User-defined Security Policies.

In *2008 IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, CA, USA, May 2008.

- [47] Nikhil Swamy and Michael Hicks. Verified Enforcement of Stateful Information Release Policies. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 21–32, Tucson, AZ, USA, June 2008.
- [48] Hiroshi Unno, Naoki Kobayashi, and Akinori Yonezawa. Combining Type-based Analysis and Model Checking for Finding Counterexamples against Non-interference. In *1st Workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 17–26, Ottawa, ON, Canada, 2006.
- [49] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.
- [50] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 32–41, San Diego, CA, USA, June 2007. ACM.
- [51] Gary Wassermann and Zhendong Su. Static Detection of Cross-site Scripting Vulnerabilities. In *30th International Conference on Software Engineering (ICSE 2008)*, pages 171–180, Leipzig, Germany, May 2008.
- [52] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *11th USENIX Security Symposium*, San Francisco, CA, USA, August 2002.
- [53] Lantian Zheng and Andrew C. Myers. Securing Non-intrusive Web Encryption through Information Flow. In *3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 125–134, Tucson, AZ, USA, June 2008.