Challenges for Information-flow Security

Steve Zdancewic
stevez@cis.upenn.edu

University of Pennsylvania, Philadelphia PA 19104, USA

1 Introduction

Protecting confidential data in computing environments has long been recognized as a difficult and daunting problem. All modern operating systems include some form of access control to protect files from being read or modified by unauthorized users. However, access controls are insufficient to regulate the propagation of information after it has been released for processing by a program. Similarly, cryptography provides strong confidentiality guarantees in open, possibly hostile environments like the Internet, but it is prohibitively expensive to perform nontrivial computations with encrypted data. Neither access control nor encryption provide complete solutions for protecting confidentiality.

A complementary approach, proposed more than thirty years ago, is to track and regulate the *information flows* of the system to prevent secret data from leaking to unauthorized parties. This can be done either dynamically, by marking data with a *label* describing its security level and then propagating those labels to all derivatives of the data, or statically, by analyzing the software that processes the data to determine whether it obeys some predefined policy with respect to the data. Arguably, a mostly static approach (perhaps augmented with some dynamic checks) is the most promising way of enforcing information-flow policies.

A recent, comprehensive survey by Sabelfeld and Myers [11] includes 147 references to publications related to information-flow security. The bulk of these papers are concerned with defining and refining variations on *noninterference*, the fundamental information-flow property that essentially requires that secret information not affect publicly observable behavior of a system. Many of the remaining papers describe approaches to enforcing information-flow policies using program analysis techniques. Yet despite this large body of literature and considerable, ongoing attention from the research community, information-flow based enforcement mechanisms have not been widely (or even narrowly!) used.

The real challenge in information-flow security is *not* in giving better, more precise definitions of noninterference and related properties for more complicated combinations of language features and system models. Nor is the real challenge implementing languages that support information-flow policies; the programming languages Jif, developed by Myers et al. [7], and Flow Caml, developed by Simonet and Pottier [12, 9], provide high-level, realistic programming languages with support for sophisticated information-flow controls. Although there are certainly interesting open questions in both its theory and implementation, the real challenge for information-flow security is demonstrating that all of this theory and these language designs are actually useful—we need to apply the technology to real problems, or, failing that, understand why such an appealing technology is not useful in practice.

The remainder of this paper examines the current status of information-flow technology and tries to identify some of the main obstacles of putting it into practice. The short list below is no doubt incomplete, and this paper provides no definitive solutions, but it is derived from experience with using Jif and it should serve as a useful starting point for future research.

2 Challenges

To understand the difficulty in applying language-based information-flow controls in practice, it is helpful to look at where similar technology *is* used. Perhaps the most widely used security mechanism related to information-flow is the "taint checking" mode for the scripting language Perl. With this feature enabled, Perl scripts tag data that arrives from untrusted sources (such as the network) and raise an error if such tainted data is passed to potentially exploitable functions (such as system calls). Because it uses a purely dynamic enforcement mechanism, Perl does not track implicit flows (those that arise due to control-flow), but it does provide a way of downgrading the data from "tainted" to "untainted" by pattern matching. Perl's security policy is implicit and not configurable, and the mechanism is unsound, but it is apparently successful in preventing many security violations. Note that Perl is not concerned with preventing secret information from being leaked. Also, taint checking only seeks to reduce vulnerabilities, not eliminate them.

Each of the following sections examines a particular challenge for deploying language-based information-flow technology. Comparison with Perl highlights the differences between current academic research and real-world practice.

2.1 Integrating information-flow controls with existing infrastructure

One significant challenge in building real applications that have information-flow policies is getting the new application to interoperate correctly with existing infrastructure. Current operating systems and software libraries are not designed with information-flow policies in mind and it is not practical to rewrite all of this existing code to account for information-flow constraints. One possibility for handling existing APIs is to provide wrapper interfaces that properly take into account the behavior of the underlying implementation, but this is almost certainly going to be conservative or unsound.

Besides backwards compatibility issues at the code level, there are other problems in getting differing security models to interoperate. Operating systems provide security abstractions like the notion of user and access control list; languages like Java and C# provide their own security abstractions like stack inspection; cryptographic techniques provide abstractions like keys and digital certificates; information-flow systems usually specify policies in terms of a lattice of security labels. Allowing all of these different security mechanisms to work together is necessary for building real applications because real applications need to do things like file and network I/O. Some progress in this area has been made: The work by Chothia et al. integrates distributed access control and PKI with a lattice model [2]. Banerjee and Nauman demonstrate how information-flow policies can be made to interact well with stack inspection [1]. Tse and Zdancewic [13] describe a means of connecting dynamic policy information as provided by an operating system with the static analysis done by type-checking. But there is still much to be done before these approaches are suitable for practical applications.

In contrast, Perl provides a simple, fixed security interface to the run-time system. It trades flexibility for ease-of-use, and the fact that it is unsound gives a great degree of freedom for interfacing with existing code.

2.2 Escaping from the confines of pure noninterference

Another significant challenge for applying information-flow techniques is that they often strive to prevent *all* information-flows from secret data to public observers, usually by enforcing a noninterference property. There are at least two problems with this approach. First, in many (if not most) applications the appropriate security policy *does* permit such downward flows. Therefore, noninterference and the like are simply not the desired policy in most cases. Second, information-flow analyses are necessarily conservative, because giving a precise characterization of information-flow reduces to the halting problem. This means that some perfectly valid (even noninterfering) programs will be rejected as insecure. These two problems imply the need for some way to specify policies that include downgrading (also called declassification for confidentiality policies, or endorsement for integrity policies). The challenge is determining what the nature of such a downgrading mechanism should be and what kinds of security guarantees it permits.

There are a number of approaches to the downgrading problem. Intransitive noninterference [10] ensures that downward information flows must pass through certain trusted system components. Volpano and Smith [14] justify certain downward flows through a restricted test operator by showing that an adversary would need time exponential in the length of the secret to learn the entire secret. The decentralized label model, developed by Myers and Liskov [5, 6], provides a form of authorization-based access control for the `declassify` operator used in Jif. Zdancewic, Myers, and Sabelfeld [16, 15, 8] later refined the model to require that the integrity of the decision to perform the declassification be sufficiently trusted to justify the downgrading. More recently, various ways of meaningfully relaxing noninterference have been studied [3, 4].

All of these approaches suffer in practice either because the effects of downgrading cannot be easily accounted for (Jif's `declassify`), or because they are too restrictive (the Volpano and Smith test operator), or because they are difficult to enforce (intransitive and relaxed noninterference).

Perl builds endorsement in to its pattern matching construct, but does not provide any guarantees that the dynamic check is sensible. For instance the "match anything" pattern can be used to remove the taint mark from an arbitrary piece of data. As for Jif's `declassify`, using Perl's pattern match construct means that the downgrading policy is essentially the program itself—to understand the policy you must understand the program. This violates the principal of separating specification from implementation.

Noninterference is not practical, and it is not known how to tractably enforce useful, well-defined alternative information-flow policies.

2.3 Managing complex security policies

A final, and perhaps most important, challenge for information-flow security stems from the difficulty of managing complex security policies. As we have seen from the above discussion, realistic policies do not fall into the simple noninterference-like models. Furthermore, systems must interact with existing security infrastructures, such as the access controls provided by an operating system. When combined, these two requirements mean that the policies themselves become quite complex. A typical program manipulates data owned by multiple principals, some of whom are known when the program is written, some of whom are not known until run time. Language-based information-flow techniques require that the annotations in the program faithfully describe the desired policy. Even for relatively small programs (say, less than 1000 lines of code), the profusion of possible policies quickly becomes bewildering. The programmer must not only understand the algorithm she is implementing, but she must also understand what the desired security policy is and how to formalize it using annotations. By contrast, Perl provides one hard-wired integrity policy that is applicable in most cases. However, such an inflexible approach is not suitable for confidentiality policies because there is no reasonably generic notion of what is a secret and who should be allowed to share it.

While there is some work related to type inference and polymorphism that may help ameliorate this problem, we do not yet have the tools to easily describe desired security policies. We do not understand the right high-level abstractions for specifying information-flow policies.

3 Conclusion

Despite their long history and appealing strengths, information-flow mechanisms have not yet been successfully applied in practice. There are a number of obstacles to using language-based techniques, among them: integration with existing security mechanisms, the inadequacy of strict noninterference, and the difficulty of managing security policies. Taking a cue from Perl's (modest?) success at using information-flow concepts in practice, perhaps it is time that the information-flow research community stop striving for the unattainable goal of noninterference. After all, perfect security is unattainable—noninterference is proved relative to some level of abstraction, which does prevent information leaks at other levels of abstraction. Instead, the challenge is to demonstrate that the information-flow techniques that have been developed over the last thirty years can be applied to practical systems to increase our confidence that they are secure.

References

1. Anindya Banerjee and David A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 155–169. IEEE Computer Society Press, 2003.
2. Tom Chothia, Domminic Duggan, and Jan Vitek. Type-based distributed access control. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 170–184. ieee, jun 2003.
3. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 186–197, Venice, Italy, January 2004.
4. Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. Submitted for publication, July 2004.
5. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.
6. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
7. Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.
8. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *csfw17*, pages 172–186. IEEE Computer Society Press, July 2004.
9. François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.
10. A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, 1999.
11. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
12. Vincent Simonet. Flow caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.
13. Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 179–193, 2004.
14. Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, January 2000.
15. Steve Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics.* Electronic Notes in Theoretical Computer Science, March 2003.
16. Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.