Programming with Explicit Security Policies

Andrew C. Myers

Cornell University
andru@cs.cornell.edu

Abstract. Are computing systems trustworthy? To answer this, we need to know three things: what the systems are supposed to do, what they are not supposed to do, and what they actually do. All three are problematic. There is no expressive, practical way to specify what systems must do and must not do. And if we had a specification, it would likely be infeasible to show that existing computing systems satisfy it. The alternative is to design it in from the beginning: accompany programs with explicit, machine-checked security policies, written by programmers as part of program development. Trustworthy systems must safeguard the end-to-end confidentiality, integrity, and availability of information they manipulate. We currently lack both sufficiently expressive specifications for these information security properties, and sufficiently accurate methods for checking them. Fortunately there has been progress on both fronts. First, information security policies can be made more expressive than simple noninterference or access control policies, by adding notions of ownership, declassification, robustness, and erasure. Second, program analysis and transformation can be used to provide strong, automated security assurance, yielding a kind of security by construction. This is an overview of programming with explicit information security policies with an outline of some future challenges.

1 The Need for Explicit Policies

Complex computing systems now automate and integrate a constantly widening sphere of human activities. It is crucial for these systems to be trustworthy: both secure and reliable in the face of failures and malicious attacks. Yet current standard practices in software development offer weak assurance of both security and reliability. To be sure, there has been progress on automatic enforcement of simple safety properties, notably type safety. And this is valuable for protecting systems from code injection attacks such as buffer overruns. But many, perhaps most serious security risks do not rely on violating type safety. Often the exposed interface of a computing system can be used in ways unexpected by the designers. Insiders may be able to misuse the system using their privileges. Users can sometimes learn sensitive information when they should not be able to. These serious vulnerabilities are difficult to identify, analyze, and prevent.

Unfortunately, current practices for software development and verification do not seem to be on a trajectory that leads to trustworthy computing systems. Incremental progress will not lead to this goal; a different approach is needed. We have been exploring a language-based approach to building secure, trustworthy systems, in which programs are annotated with explicit, machine-checked information security policies relating to

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 1–4, 2005.

properties such as the confidentiality and integrity of information. These properties are both crucial to security and difficult to enforce. It is possible to write relatively simple *information flow* policies that usefully capture these aspects of security. These explicit policy annotations then support automatic enforcement through program analysis and transformation.

2 Limitations of Correctness

Of course, the idea of automatic verification has always been appealing—and somewhat elusive. The classic approach of verifying that programs satisfy specifications can be a powerful tool for producing reliable, correct software. However, as a way to show that programs are secure, it has some weaknesses. First, there is the well-known problem that the annotation burden is high. A second, less appreciated problem is that classic specifications with preconditions and postconditions are not enough to understand whether a program is secure. Correctness assertions abstract program behavior; if the abstraction leaves out security-relevant information, the actual program may contain security violations (especially, of confidentiality) invisible at the level of the abstraction. Thus, it's also important to understand not only what programs do but also what they *don't* do. Even if the program has no observable effect beyond what its specification describes, the specification itself may allow the confidential information to be released. A third problem is that correctness assertions don't address the possible presence of malicious users or code, which is particularly problematic for distributed systems.

3 End-to-End Information Security

If classic specification techniques are too heavyweight and yet not expressive enough, what are the alternatives? One possibility is information flow policies, which constrain how information moves through the system. For example, a policy that says some data is confidential means that the system may not let that data flow into locations where it might be viewed insecurely. This kind of policy implicitly controls the use of the data without having to name all the possible destinations, so it can be lightweight yet compatible with abstraction. Furthermore, it applies to the system as a whole, unlike access control policies, which mediate access to particular locations but do not control how information propagates. One can think of information flow policies as an application of the end-to-end principle to the problem of specifying computer security.

Information flow policies can express confidentiality and integrity properties of systems: confidentiality is about controlling where information flows to; integrity is about controlling where information flows from. Integrity is also about whether information is computed correctly, but even just an analysis of integrity as information flow is useful for ensuring that untrustworthy information is not used to update trusted information.

Fundamentally, information flow is about dependency [ABHR99], which makes sense because security cannot be understood without understanding how components depend on one another. The approach to enforcing information flow that has received the most attention is to analyze dependency at compile time using a *security type sys-*

tem [SM03]. The Jif programming language [Mye99], based on Java, is an example of a language with a type system for information security.

The other appealing aspect of information flow policies is that they can be connected to an underlying semantic security condition, noninterference. Noninterference says roughly that the low-security behavior of a system does not change when high-security inputs are changed. This condition (which has many variants) can be expressed in the context of a programming language operational semantics [VSI96], making possible a proof that a security type system constrains the behavior of the system.

4 Whole-System Security and Mutual Distrust

Many of the computing systems for which security is especially important are distributed systems serving many principals, typically distributed at least partly because of security concerns. For example, consider a web shopping service. At the least, it serves customers, who do not entirely trust the service, and the companies selling products, who do not trust the customers or each other. For this reason, the computational resources in use when a customer is shopping are located variously on the customer's computer, on the web service provider, and on the seller's computers. It is important to recognize that these principals have their own individual security requirements; the system as a whole must satisfy those requirements in order for them to participate.

To enforce information security for such a system, it is necessary to know the requirements of each of the principals. The decentralized label model [ML00] is an information flow policy language that introduces a notion of information flow policies owned by principals. For example, in the context of confidentiality, a policy $p_1 : p_2$ means that principal p_1 owns the policy and trusts principal p_2 to read the corresponding information. More generally, p_1 trusts p_2 to enforce the relevant security property on its behalf. This structure makes it possible to express a set of policies on behalf of multiple principals while keeping track of who owns (and can relax) each policy.

For example, suppose we are implementing the game of Battleship with two players, A and B. Player A wants to be able to read his own board but doesn't want B to read it, so the confidentiality label is $\{A : A\}$. For integrity, both principals want to make sure that the board is updated in accordance with the rules of the game, so the integrity label has two owned policies: $\{A : A \wedge B,\ B : A \wedge B\}$, where $A \wedge B$ is a conjunctive principal representing the fact that both A and B must trust the updates to A's board.

5 Security Through Transformation

Secure distributed systems achieve security through a variety of mechanisms, including partitioning code and data (as in the web shopping example), replication, encryption, digital signatures, access control, and capabilities. Analyzing the security of a complex system built in this fashion is currently infeasible.

Recently, we have proposed the use of automatic program transformation as a way to solve this problem [ZZNM02]. Using the security policies in a non-distributed program, the Jif/split compiler automatically partitions its code and data into a distributed system that runs securely on a collection of host machines. The hosts may be trusted to varying

degrees by the participating principals; a partitioning is secure if policies of each principal can be violated only by hosts it trusts. The transformation employs not only partitioning, but also all of the distributed security mechanisms above to generate distributed code for Jif programs. For example, given the labels above, Jif/split can split the code of a Battleship program into a secure distributed system.

6 Conclusions and Future Challenges

The ability to provably enforce end-to-end security policies with lightweight, intuitive annotations is appealing. Using policies to guide automatic transformation into a distributed system is even more powerful, giving a form of security by construction. However, research remains to be done before this approach can be put into widespread use.

Noninterference properties are too restrictive to describe the security of real-world applications. Richer notions of information security are needed: quantitative information flow, policies for limited information release, dynamic security policies [ZM04], and downgrading policies [CM04]. End-to-end analyses are also needed for other security properties, such as availability.

Checking information flow policies with a trusted compiler increases the size of the trusted computed base; techniques for certifying compilation would help.

The power of the secure program transformation technique could be extended by employing more of the tools that researchers on secure protocols have developed; secret sharing and secure function computation are obvious examples.

Strong information security requires analysis of how programs use information. Language techniques are powerful and necessary tools for solving this problem.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, October 2004.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[SM03] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust*, August 2004.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.