Manifest Security

Karl Crary Robert Harper Frank Pfenning
Carnegie Mellon University

Benjamin C. Pierce Stephanie Weirich Stephan Zdancewic
University of Pennsylvania

January 2007

Project Summary

This project proposes *manifest security* as a new architectural principle for secure extensible systems. Its research objectives are to develop the theoretical foundations for manifestly secure software and to demonstrate its feasibility in practice.

Manifest security applies to *extensible software platforms*—software systems that can be customized by installing third-party *extensions*. The goal of manifest security is to address two fundamental problems in this domain, both stemming from the need to protect the platform from untrusted and potentially malicious extensions. Useful software extensions often require access to system resources or sensitive information, yet permitting unrestricted access opens the possibility for abuse. It is therefore necessary, first, to *specify* policies about what resources an extension may use and how it can handle sensitive data; second, the platform must also include an effective mechanism for *enforcing* such policies. The critical components missing from existing architectures are thus (1) a general, practical means for users to specify security policies about how extensions are permitted to behave, and (2) a way of determining whether a given extension (which may be malicious) actually meets the desired policy. Manifest security addresses both of these issues.

Our formulation of manifest security introduces a novel high-level logical specification language, encompassing both authorization properties (to deal with access control) and information-flow properties (to protect confidential information and restrict the use of sensitive data released to the extension). Adherence to specifications is enforced by a combination of static and dynamic methods, and trustworthiness of the code is established by the *explicit* representation and verification of formal proofs. Cryptographic primitives form the substrate upon which the enforcement mechanisms are implemented, and they are coherently integrated into the semantics of the programming language used to implement extensions.

The primary research activities to be undertaken during the proposed four-year effort are:

- Developing a *formal security logic* that permits mathematically precise formulation of security policies, including access control and information flow properties. This logic can be used to ensure that formal security policies are consistent with informal security expectations, thereby enabling tools for *security policy analysis and certification*.
- Designing, implementing, and disseminating a *proof-carrying run-time system* and a *security-typed programming language* in which to build manifestly secure applications. The run-time system will be built on the Firefox JavaScript API using proof-carrying authorization to enforce access control restrictions. The security-type system will incorporate proofs in security logic so that compliance with the security policy can be ensured by a combination of proof checking and type checking.
- *Formalizing and analyzing* the metatheory of the security logic and the security-type system for the programming language using the *Twelf logical framework*. Metatheoretic properties include the consistency of the security logic, and safety and non-interference properties of the programming language.
- *Evaluating* the usability of these technologies by constructing and disseminating several extensions, such as a password manager extension for Firefox, written in the security-typed programming language.

Intellectual merit. Realizing manifest security requires a novel synthesis of techniques from logic, programming languages, and cryptography. The activities outlined above will advance the understanding of formal security policies and their enforcement, push the boundaries of mechanized reasoning about computer infrastructure, and lead to a new, more secure programming model.

Broader impact. Because extensible systems are in widespread use (*e.g.*, in web browsers, office software, media players, games, virtual communities, and operating systems) the concept of manifest security has significant potential for broad impact. Rigorous verification methods based on logic and type theory are increasingly important to the software industry; the project advances the use of these methods to ensure security. The participants have an established record of fostering education in the field through writing textbooks, developing new classes and course materials at their universities, and organizing summer schools for students throughout the world. The project will also employ undergraduate researchers through direct funding and the NSF Research Experience for Undergraduates program. Both participating departments have vibrant organizations supporting and promoting women in computer science, and we will work toward involving women in our project at both the undergraduate and graduate level.

1 Overview

Extensible software platforms—software systems that can be customized or modified by installing third-party *extensions*—are widely used. Familiar examples include web browsers, media players, office software, e-mail clients, online games, and virtual communities; even operating systems are commonly extensible via device drivers and custom kernel modules.

Although web browsers, games, and operating systems are substantially different kinds of software, a common fundamental tension in their design is the need to protect against malicious code while still allowing third-party developers to contribute useful extensions. The problem is that for an extension to do useful work, it might need access to resources that can potentially be misused; a malicious extension could potentially be very dangerous. For example, a password manager extension for a web browser might be given access to sensitive passwords stored on the local machine and also be permitted to transmit them over the network. Such an application would have sufficient capabilities to violate privacy restrictions, yet a blanket prohibition of access to sensitive data or to the network would render it unusable. As another example, DRM software might require access to both local files and the network to download music while enforcing usage restrictions.

To protect against malicious extensions, the security architectures for these kinds of extensible systems have adopted a number of defensive strategies. One defense is the use of bytecode interpreters and virtual machines to isolate extension code from the rest of the system. Examples include the Java Virtual Machine (JVM) [76], Microsoft's Common Language Runtime (CLR) [52], the Firefox JavaScript interface [53], and scripting languages for virtual communities such as Second Life [72]. Another approach, used by JavaScript 1.1 and Perl, is to enable a "taint checking mode" that tags data from untrusted sources and prevents it from propagating to security-sensitive operations such as system calls. These techniques protect against low-level problems like buffer overflows and format string attacks, but they do not address higher-level policy issues like the access-control problem for the password manager.

Another defense against security violations is to cripple the capabilities of extensions to ensure that they cannot do anything malicious—for example, by using run-time checks to deny access to file system or networking services. While this can be a useful strategy in many situations, it also imposes severe limits on what can be done in an extensible architecture. A better alternative is to identify "trustworthy" extensions by using cryptographic techniques to authenticate the application code; signed code might be granted more capabilities than unsigned code, presumably because the signer of the code could (at least in principle) be held accountable for problems caused by the extension. Both the JVM and the CLR support code signing for privileged code.

When combined, these methods go a certain distance towards increasing confidence in security applications, but they are clearly not enough. The Sony DRM fiasco described by Halderman and Felten [36] is one good example of what can go wrong, and there are numerous other examples of extension vulnerabilities (in plugins, device drivers, etc.) to be found in the US-CERT Vulnerability Notes Database [79].Even with

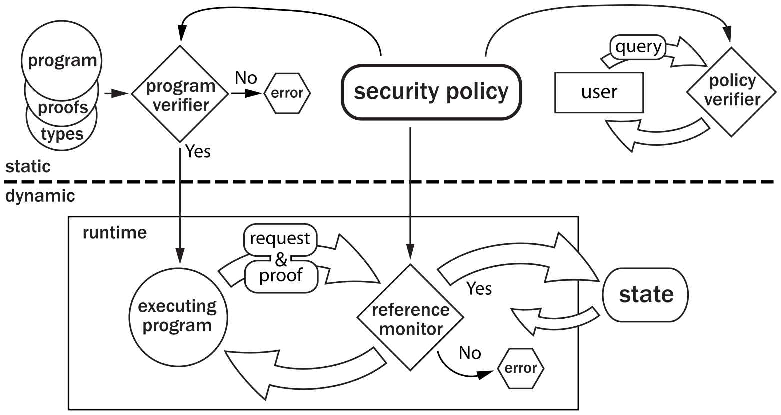

Figure 1: Manifest Security Architecture

the best intentions, extensions developed using the current technology can malfunction, leak private data, or corrupt resources in ways unanticipated by either the developer or the end user. Moreover, the end user has little control over security policies—the current means of specifying security policies are ad-hoc and informal and there is no way to check whether a given extension actually complies with the desired policy, even if the vendor claims it does! The critical components missing from the security architectures of extensible systems are (1) a general, practical means for users to specify security policies about how extensions are permitted to behave, and (2) a way of determining whether a given extension (which may be malicious) actually meets the desired policy. These fundamental problems are what we intend to address with this research project.

Manifest security. We propose a radically new approach to secure extensible system architectures that begins instead from high-level logical specifications, encompassing both authorization properties (to deal with access control) and information-flow properties (to protect confidential information and restrict the use of sensitive data released to the extension). Adherence to specifications is enforced by a combination of static and dynamic methods, and trustworthiness of the code is established by the explicit representation and verification of formal proofs. Cryptographic primitives form the substrate upon which the enforcement mechanisms are implemented, and they are coherently integrated into the semantics of the programming language. In our approach, security properties are made manifest in formal proof objects, just as the truth of a mathematical theorem is manifest in its proof. We therefore refer to our approach as *manifest security.*

The architecture of a manifestly secure framework is depicted in Figure 1. The central concept is the security policy, which consists of a collection of axioms written in a formal security logic. Authorization and information flow properties are expressed by formal proofs in security logic that may be mechanically checked for validity. Security policies are themselves subject to precise analysis and verification to ensure that they capture informal security requirements. Reference monitors for security-sensitive resources require a formal proof that access is permissible; a proof checker for security logic embedded in the resource monitor ensures the validity of the proof. Programs in a security-typed language contain evidence of compliance with the security policy in their types, so that they may be verified prior to execution to ensure that the policy is not violated.

There are many benefits to manifest security. One is a clear separation of policy from implementation, with an enforceable relationship between the two in an actual running system. Another benefit is scalability and practicality—just as type-safe languages provide a scalable, practical way for programmers to rule out buffer overflows and memory errors, the type system used in a manifestly secure language will give programmers a practical way to construct secure software. Furthermore, the formal, logical expression of policies provides an easy path towards their mechanical analysis, verifying intended consequences of policy specification and ruling out unintended ones. Finally, manifest security is expressive, flexible, and open-ended, inheriting these properties from its foundations in logic and formal proof.

Realizing manifest security requires fundamental advances at the interfaces between logic, cryptography, and programming languages. The expected contributions are:

1. We will develop *formal security logics* that permit mathematically precise formulation of security policies, including access control and information flow properties. To ensure that formal security policies are consistent with informal security expectations, we will develop logical techniques for *security policy analysis and certification.* Our plans are described in more detail in Section 2.1 below.

2. We will develop a *proof-carrying run-time system* and a *security-typed programming language* in which to build manifestly secure applications. As a testbed for evaluation and dissemination of real extensions, we will deploy this run-time system and programming language in the context of the Mozilla/Firefox web browser. The run-time system will be built on the Firefox JavaScript API, using proof-carrying authorization [10, 12] to enforce access control restrictions. The security-type system will incorporate proofs in security logic so that compliance with the security policy can be ensured by a combination of proof checking and type checking. More details are given in Section 2.2.

3. We will *formalize and analyze* the metatheory of the security logic and the security-type system for the programming language. This includes properties such as the consistency of the security logic, and safety and non-interference properties of the programming language. We will use the *Twelf logical framework* to formalize the logic and language, and mechanically to verify its metatheory. See Section 2.3.

4. We will *evaluate* our results by rewriting existing Firefox extensions in our security-typed programming language and constructing our own from scratch; planned extensions include a password manager and a "web recorder" and bookmark manager. We will *disseminate* these extensions and the other software artifacts we construct (such as the extension platform itself and our security policy analysis tools) under an open-source license. An additional mode of evaluation will be using our logic and extension platform in teaching both undergraduates and graduate students from outside the project. See Section 2.4.

We assume, throughout, that we can trust the machine on which our software platform is running (the hardware and OS, the implementation of the web browser we are extending, etc.). However, we do not trust the developers or distributors of extensions: we assume that an attacker can present arbitrary programs in our security-typed language, which we must execute if they pass our verifier. Moreover, we assume that the attacker controls the network and most remote hosts, but that certain designated hosts can be trusted, as long as we communicate with them securely. This enables more interesting extensions. Section 2.5 offers additional comments about the attack model we have in mind and lists some issues that are left outside the scope of the proposed research.

For the sake of focus, we are concentrating our implementation efforts on developing a secure extension architecture for a particular web browser; this will allow us to demonstrate and disseminate concrete results during the four year grant period However, we expect the fundamental contributions of the research will be broadly applicable to extensible software architectures.

2 Proposed Work

Our approach to building application extensions with manifestly secure access to private information is founded on logic, type theory, and mechanically verifiable proofs. This section discusses in more detail the technical challenges we must face in each of these areas and our plans for addressing them. For the sake of exposition, this section divides the work into several distinct strands, but in practice these strands will be highly interdependent: we expect to pursue all the strands in parallel, at both project sites, with all of the PIs contributing (to a greater or lesser extent) to every activity.

2.1 Security Policies: A Linear Logic of Authorization and Knowledge

The crux of manifest security is the use of a *formal security logic* to express security policies and verify compliance with them. The *assertions*, or *judgments*, of the logic make claims about principals and resources, including *access control* judgments, such as "principal A may access resource R", and *information flow* judgments, such as "principal A may know information I." (Here principals refer to cryptographic keys obtained at run time, and resources are references to run-time data structures carrying information.) A *security policy* consists of a *logical theory*, a collection of axioms and inferences that specify in declarative form the conditions under which access control and information flow judgments may be derived. Security restrictions are enforced by a combination of dynamic and static methods. Dynamically, a reference monitor embedded in our run-time system requires a *formal, mechanically checkable proof* of an appropriate assertion of the security logic. the reference monitor employs a proof checker to verify that the purported proof is valid according to the security policy. However, as we shall argue in Section 2.2, we can sometimes verify statically—before the program is executed—that such a proof will exist, in which case we do not need to check it at run time.

We begin by decomposing the problem into authorization and information flow. *Authorization* answers the question of which principals are permitted to access which resources. *Information flow* specifies the permissible consequences of properly authorized access.

Authorization policies. We want a logic in which one can reason about whether a principal should have access to a resource. Logics for reasoning about access control go back to work by Abadi et al. [6, 2]. However, prior work does not completely satisfy our design criteria—in particular, generality and extensibility are difficult to combine with the ability to reason mechanically about policies as a whole (see the related work discussion below).

Briefly, a principal K should be granted access to a resource R exactly if there is a proof of $\mathsf{may\text{-}access}(K, R)$. We may understand the meaning of this proposition by considering the pertinent judgments and proof rules [30, 49, 63]. The most basic judgment is that of the truth of a proposition, written as A *true*. We furthermore need a judgment of *affirmation*, written K *affirms* A, expressing a policy of K. For example, K *affirms* $\mathsf{may\text{-}access}(L, R)$ is a policy statement by principal K that L may access resource R. We say that K controls resource R if K *affirms* $\mathsf{may\text{-}access}(L, R)$ implies the *truth* of $\mathsf{may\text{-}access}(L, R)$. Therefore, control of a resource is expressible as a logical implication. The final ingredient is the standard notion of hypothetical judgment: $\Gamma \Longrightarrow A\ \mathit{true}$ or $\Gamma \Longrightarrow K\ \mathit{affirms}\ A$, where Γ is a collection of assumptions of the form B *true* or K *affirms* B.

We now sketch a sequent calculus for reasoning about authorization. We begin with the so-called judgmental rules, which explain the meaning of the judgments:

$$\frac{}{\Gamma, P\ \mathit{true} \Longrightarrow P\ \mathit{true}} \qquad\qquad \frac{\Gamma \Longrightarrow A\ \mathit{true}}{\Gamma \Longrightarrow K\ \mathit{affirms}\ A}$$

The first rule expresses that from the assumption P we can obtain the conclusion P; the second, that when A *true* any principal K is prepared to affirm A. (Since A is true and has an explicit proof, there is no reason for K to deny it.) Conversely, if K *affirms* A, then A is true from K's point of view—i.e., we may assume that A is true while establishing an affirmation for the same principal K:

$$\frac{\Gamma, A\ \mathit{true} \Longrightarrow K\ \mathit{affirms}\ C}{\Gamma, K\ \mathit{affirms}\ A \Longrightarrow K\ \mathit{affirms}\ C}$$

In order to use affirmations within propositions (to form policies that require the conjunction of two affirmations, for example), the logic must internalize them as propositions. The syntax $\langle K\rangle A$ packages an affirmation judgment as a proposition:

$$\frac{\Gamma \Longrightarrow K\ \mathit{affirms}\ A}{\Gamma \Longrightarrow \langle K\rangle A\ \mathit{true}} \qquad \frac{\Gamma, A\ \mathit{true} \Longrightarrow K\ \mathit{affirms}\ C}{\Gamma, \langle K\rangle A\ \mathit{true} \Longrightarrow K\ \mathit{affirms}\ C}$$

An authorization policy is now just a set of assumptions Γ. An authorization query is a conclusion, usually of the form K *affirms* $\mathsf{may\text{-}access}(L, R)$ where K controls resource R. Principal L will be granted access if there is a proof of the query from Γ. Our authorization architecture will follow proof-carrying authorization [10, 12], wherein L supplies such a proof explicitly for validation by a reference monitor implemented in the run-time system. At the leaves of these proofs are digitally signed certificates that witness the policy statements of the principals as collected in Γ.

All this raises several issues, such as how to concretely express policies and proofs, how to assemble proofs, and how to verify their correctness. Our architecture will use a logical framework [64] that is explicitly designed for the representation of logics and proofs. This design makes the architecture inherently open-ended: we can enrich our logic with further connectives while still using the same implementation. Furthermore, we can formally reason about the logic and about specific policies using the meta-theoretic reasoning capabilities of the framework. This is useful, for example, to establish that a security policy is consistent.

Information-flow policies. Authorization policies govern which principals are allowed to access which resources, but they do not specify what those principals may do with the data once they have permission to access it. Information-flow policies, in contrast, restrict the propagation and dissemination of information. We capture this logically by specifying in the policy what principals *may know*. In other words, we need to develop an epistemic logic appropriate for this setting.

It is well-known that, in general, information flow is impossible to accurately verify dynamically, essentially because it is a property of all executions of a program rather than a single one [50]. In the primary implementation scenario we are envisioning—extensions written in our language for an extensible web browser platform—we can analyze programs statically to check information flow properties against the policy specification. This is complemented by some dynamic checking, obtaining overall guarantees from a combination of the two. For example, in a password manager we may wish to verify that the stored password P for a given site (represented as a principal S) will only be posted to that particular site. The post request is accompanied by a formal proof that S may know P.

In order to define a logic of knowledge, we need a new judgment, K *knows* A, where A is a proposition. From a policy perspective, it means that K *may know* A; as an assumption we read it as "if K knows A then ...". What are the intrinsic (that is, policy-independent) logical properties of knowledge? Clearly, if K knows A, then A should be true. Consequently, any judgement J entailed by A *true* is entailed by K *knows* A (below, left rule):

$$\frac{\Gamma, A\ \mathit{true} \Longrightarrow J}{\Gamma, K\ \mathit{knows}\ A \Longrightarrow J} \qquad\qquad \frac{\Gamma|_K \Longrightarrow A\ \mathit{true}}{\Gamma \Longrightarrow K\ \mathit{knows}\ A}$$

The converse is false, and this is the very essence of secrecy: there are many true propositions that K does not (and should not) know. We establish that K may know A by showing that K can infer A using only its own knowledge. We formalize this using the restriction operator $\Gamma|_K$, which erases from Γ all hypotheses *not* of the form K *knows* B (above, right rule).

As before, we can internalize the judgment K *knows* A as a proposition, written $[\![K]\!]A$:

$$\frac{\Gamma \Longrightarrow K\ \mathit{knows}\ A}{\Gamma \Longrightarrow [\![K]\!]A\ \mathit{true}} \qquad\qquad \frac{\Gamma, K\ \mathit{knows}\ A \Longrightarrow J}{\Gamma, [\![K]\!]A\ \mathit{true} \Longrightarrow J}$$

State and stateful policies. At this point the logic can specify and reason about authorization and its information flow consequences. However, the logic as we have described it so far is monotonic: during a proof we can establish more affirmations and infer additional knowledge for the principals, but we can never take away knowledge. Consequently, the system can not model consumable resources, nor systems with essential state changes. In order to capture such systems, it is necessary to move to a linear logic [31]. This has recently been used in the security domain to represent one-time permissions in various forms [11, 28, 16].

Space permits only the briefest sketch of the logical system with linearity that we have in mind. We distinguish between persistent assumptions (including all the ones discussed above) and linear assumptions, which must be used exactly once in a proof. Such assumptions can model either consumable, one-time certificates (for linear affirmations), modifiable data, or the *possession* of a resource (for linear knowledge). For further background and discussion on a linear logic including both affirmations and knowledge, see [28].

An example policy. To illustrate some of these ideas, consider a simplified version of the password manager example. The principals are the user U, the executing password manager, the sites (URLs) S to which the passwords apply, and some secure, persistent storage F.

A first version of the policy does not allow the user to change the password:

- If U affirms that its id and password for URL S are I and P, and if the password hint is P', then the filesystem F may know this information (presumably the password is stored in encrypted form, so no one else can learn this):

$\langle U\rangle \mathsf{id\_and\_passwd}(S, I, P, P') \supset [\![F]\!]\langle U\rangle \mathsf{id\_and\_passwd}(S, I, P, P')$

- If the filesystem has stored U's password information for site S, then the extension may post this information to S (presumably along a secure channel):
  $[\![F]\!]\langle U\rangle \mathsf{id\_and\_passwd}(S, I, P, P') \supset [\![S]\!]\mathsf{password}(I, P)$
- U can see its id and password hint and the length of the password, but not the password itself:
  $[\![F]\!]\langle U\rangle \mathsf{id\_and\_passwd}(H, N, P, P') \mathbin{\&} \mathsf{length}(P, K) \supset [\![U]\!]\mathsf{id\_and\_hint}(H, N, K, P')$

Using linearity, we can refine the above policy so that it that permits the user to change passwords. The policy clause below requires that the user confirm the old password:

$$\begin{array}{rl} & [\![F]\!]\langle U\rangle \mathsf{id\_and\_passwd}(S, I, P, P') \otimes \langle U\rangle \mathsf{old\_id\_and\_passwd}(S, I, P) \\ \otimes & \langle U\rangle \mathsf{new\_id\_and\_passwd}(SN, IN, PN, PN') \multimap [\![F]\!]\langle U\rangle \mathsf{id\_and\_passwd}(SN, IN, PN, PN') \end{array}$$

Policy management. The proposed logic provides considerable expressive power to define appropriate policies for various types of applications, and the proposed work on policy analysis (Section 2.3) will provide a rigorous basis by which to analyze the consequences of policies. However, it is unrealistic to expect that a typical user of the system will be able to utilize these tools directly. In practice, policies will be devised and analyzed by the cognoscenti, which raises the natural question: how can typical users choose good policies?

We cannot solve this problem through a collection of default policies, for two reasons. First, the universe of possible extensions is open-ended. Each class of applications (*e.g.,* password manager, tax preparer, etc.) requires a distinct policy, and and one clearly cannot provide, a priori, an appropriate policy for every such class that will ever be conceived. Second, for some classes of applications, it may be controversial what the appropriate policy is. For example, there will likely be considerable controversy over what privileges should be afforded to DRM software or other "benign spyware."

We envision a solution reminiscent of PGP's web of trust model [83]. A user will specify a person whom he or she trusts to devise appropriate security policies. That person devises a collection of policies and digitally signs them. When the extension manager needs to know the policy for an application class, it can automatically fetch the policy and verify its digital signature, before checking that the application satisfies the policy.

In fact, the web of trust can be represented entirely within the logic, with no need for external mechanisms. For example, if A trusts B to devise appropriate policies for the application class C, we can state the meta-policy:

$$\forall P \in C.\, \langle B\rangle \mathtt{permitted}(P) \supset \langle A\rangle \mathtt{permitted}(P)$$

This meta-policy states that if a program P belongs to the class C, and B affirms it should be permitted to execute, then A also affirms it is permitted to execute.

Contributions. The main contributions of this thread of research may be summarized as follows: (1) We will design and implement a policy logic incorporating affirmation (for reasoning about authorization), knowledge (for reasoning about information flow), and linearity (for consumable authorities and resources), combining them into a coherent foundation for security policy specification. (2) Following the philosophy of manifest security, we will formalize the meta-theoretic properties of the policy logic, including cut elimination and various forms of policy analysis, such as noninterference. In this logic, noninterference theorems take the form $(\Gamma, K \textit{ affirms } A \Longrightarrow L \textit{ affirms } C)$ *if and only if* $(\Gamma \Longrightarrow L \textit{ affirms } C)$, under various circumstances—for example, when Γ does not mention K in a negative position and K is distinct from L. This means K cannot interfere with authorization for L. Part of the novel contribution here will be connecting this characterization of noninterference to more standard formulations found in the programming languages literature. (3) We will develop appropriate cryptographic enforcement mechanisms for the linear security logic to account for the presence of consumable certificates and resources. (The corresponding problem without linearity is relatively well understood: a statement of the form K *affirms* A is either directly a digital certificate with contents A signed by a key corresponding to principal K or else a chain of formal proof steps ultimately relying on such signed certificates.)

Related Work. Since Abadi *et al.*'s seminal work [6], there have been numerous proposals for authorization logics [40, 15, 20, 2, 46, 45, 68, 3]. Many of these have different aim and scope from our work in that they are designed to capture and reason about existing mechanisms, rather than being based on purely logical principles. Also, as far as we are aware, none (besides our own preliminary research [29, 28]) have been investigated from the meta-theoretic perspective to prove, for example, cut elimination and the noninterference theorems that follow from them. Moreover, prior proposals do not integrate reasoning about knowledge or consumable resources. Another line of related work explores the use of authorization logic for policy enforcement via explicit proof objects [10, 12, 13, 14] (so-called *proof-carrying authorization*) with first steps at exploring the value of linearity to model consumable credentials [11, 28, 16]. Our proposed research directly builds on this, but takes it significantly further by incorporating knowledge (and thereby information flow), possession (as linear knowledge), as well as integrating the logic into a complete programming language as described in the next section. Moreover, much of the past work on proof-carrying authorization has been carried out in the framework of classical higher-order logic or impredicative type theory, which is inherently difficult to reason about; our approach will be predicative and constructive. There is also prior research on using logics of knowledge to specify information-flow policies [60], but that work concentrates mainly on explaining the relationships among different policies and does not consider the interaction of information flow and authorization. Another approach is to use a type system for authorization as done, for example, in the KLAIM system [23]. Manifest security extends this to a much richer and more open-ended policy language at the cost of more complex enforcement mechanisms.

2.2 A Programming Language for Manifest Security

As a concrete demonstration of manifest security, we propose to build a secure extension architecture consisting of two major components: a *proof-carrying run-time system* and a *security-typed programming language*.

Proof-carrying run-time system. As suggested in Figure 1, the run-time system uses proof-carrying authorization (PCA) [10, 12] to control access to sensitive resources such as files and communication channels. Access control to such resources is based on the presentation of a formal proof of authority according to the security policy (as described in Section 2.1). The reference monitor uses a proof checker—which can be shared among all resources—that is parameterized by the governing security policy to verify the validity of the proof. For example, a reference monitor for a file F may require of a principal K a proof of $\mathsf{may\text{-}access}(K, F)$ as a condition for satisfying a read request.

To permit construction of useful applications, we will build the secure run-time system as a layer on top of the Firefox JavaScript API. Doing so provides access to the browser context (including cookies, caches, and passwords), as well as access to a user interface, the ambient file system, and the network. Importantly, the JavaScript API also provides standard cryptographic services, such as signed digital certificate management, which are essential for our work. Augmenting this API with proof-carrying authorization provides us with an access control mechanism that is parameterized by a declarative security policy written in our security logic. This permits us to experiment with a variety of applications and policies without changing the underlying run-time. We plan to develop a JavaScript library for the proof-carrying API so that we can build proof-concept applications at an early stage, and to support development of a security-typed programming language.

Security-typed programming language. The proof-carrying run-time system is sufficient to build extensions that respect the access control restrictions imposed by the security policy. However, this means that proofs must be constructed at run-time, and that any errors in the application will only be noted at execution time, rather than development time. We would prefer to have a language that permits tracking of dynamic proof obligations at compile time to the extent possible. We may also wish to assign confidentiality or integrity levels to resources and principals, and impose the requirement that confidential data cannot reach untrusted principals. Such a restriction amounts to a proof of the absence of a flow of knowledge from one principal to another. A static approach to policy enforcement is now necessary, because run-time methods cannot, in general, be used to enforce information-flow restrictions—information flow is a property of the set of all possible executions of a program [50], and aborting an access may itself reveal information. These considerations motivate the development of a *security-typed programming language* that tracks at compile time the information flow and access control obligations of application code. This language will be executed on the secure run-time system, either by interpreting it or by compiling it to JavaScript with calls to the proof-carrying API.

To reconcile static policy enforcement with dynamic authorization and policy manipulation, the type system of our language must statically guarantee that run-time authorization proofs are correctly assembled and supplied. For example, programs written for the extensible architecture may construct proofs in the security logic at run-time in order to access sensitive resources. These proofs are presented to a reference monitor as evidence of authority to perform a sensitive operation. However, the consequences of such sensitive operations must be statically tracked to enforce information-flow policies. Furthermore, access to additional resources with the same authorization policy should be allowed without dynamic reauthorization. To verify statically that these programs comply with host policies requires that the construction of proofs in the security logic be reflected into the static type system, so that type checking involves proof checking. Consequently, type checking ensures that the run-time checks required to implement the security policy are properly performed.

The interplay between static and dynamic methods in the security-typed programming language may be illustrated by two key examples. First, suppose that the reference monitor for a resource, R, requires of a principal, K, a proof of the proposition $\mathsf{may\text{-}access}(K, R)$, in order for K to access R. To track the run-time requirement of the reference monitor in the type system, we assign the following type to the read primitive associated with resource R:

$$\forall K{::}\mathsf{Principal}.\forall P{::}\mathsf{Proof}(\mathsf{may\text{-}access}(K, R)).\mathsf{string} \mathbin{@} K$$

This type states that for a principal K to access R it must present the required proof, and that the resulting string (say) is controlled by the principal K. This means that the string was computed on behalf of K, and may only be used by K, or those K' authorized to act on K's behalf. Assigning "control" of a value permits us to track and enforce information flow constraints. For example, principal hierarchies can be expressed by logical implications of the form $\mathsf{may\text{-}access}(K, R) \supset \mathsf{may\text{-}access}(K', R)$. Possession of a proof of such a proposition amounts to delegation of authority to access R from K to K'. Similar methods may also be used to model robust declassification schemes [82, 56].

The proofs required by the type system are constructed from the security policy, together with "primitive proofs" that arise from the acquisition of credentials at run time. This can be neatly modeled in the security-typed language by a primitive of the form

$$\begin{array}{l} \mathsf{case\ dynamicallyAuthorized\ may\text{-}access}(K, R)\ \{ \\ \qquad \mathsf{ok}(u :: \mathsf{may\text{-}access}(K, R)) \Rightarrow e_1 \\ \qquad \mid \mathsf{fail} \Rightarrow e_2 \\ \} \end{array}$$

which, at run-time, seeks authority for K to access R using a conventional security protocol, such as a password check. If the attempt succeeds, control passes to e_1 in the scope of a variable u representing a *proof* that K may access R. This proof may be used in e_1 to validate calls to a reference monitor for R, as described above. If the attempt fails, control passes to e_2 without such a proof, which must somehow recover from the failure. In this manner dynamic checks based on conventional cryptographic methods give rise to proofs that are used in our security-typed programming language.

What makes the design of such a language challenging, is that, for such policy queries to be useful, the static analysis of the program must take into account the results of the query. Thus, `dynamicallyAuthorized` cannot simply be a library routine: it interacts with typechecking in nontrivial ways, as the branch e_1 above is type checked under the (static) assumption $\mathsf{may\text{-}access}(K, R)$. There is a large design space here that trades off flexibility of the language with feasibility of typechecking. One issue is how "first-class" the query arguments, like $\mathsf{may\text{-}access}(K, R)$, should be. Another question is whether programs should be able to issue new policy statements, and, if so, how to represent and control this dynamic variation of authority. Finally, we need to ensure that these dynamic checks themselves cannot be used as information channels for implicit flows. We have already begun to explore this design space; see, for example, the work by Tse and Zdancewic [77, 78], which addresses some of these concerns in a domain where the security policy logic is quite restricted. The authorization logic proposed here can be seen as a natural generalization of that special case.

Proofs in the security logic are, in this manner, integrated with types, so that type checking involves proof checking. This means, however, that the programmer is responsible for constructing the required proofs, which may be tedious in practical situations. For this reason we anticipate that it will be necessary to develop a type inference algorithm that integrates some forms of proof search so that proofs may, in simple cases, be found automatically.

To complete the story, we must connect the type system of the programming language to the policies expressed in the logic described in Section 2.1. In previous work on security type systems [70], the information-flow policy that programs satisfy is embedded in the types of the program itself: One must examine types of the top-level program, APIs, global variables and other forms of I/O to determine what sort of information-flows are permitted or prohibited.

In our approach, we propose to use the logic of knowledge described previously to specify information-flow policies separately from the source code of an extension. We do so by ensuring that the type of each computation describes the information-flows effects that may occur during its execution. If only permissible information flows appear in the effects of an extension, i.e. if the information-flow policy entails all of the effects that may be caused by the extension, then we know that the extension is secure.

To compute these effects, we ascribe a type signature to the proof-carrying API that mentions statements drawn from the security logic. These types must accurately describe any information flows that occur. Returning to the password manager example from before, we might have a `post` function corresponding to the HTTP "post" command that sends data to a server at site S. Its type schema might be: `post` : $\mathsf{Data}(p) \xrightarrow{[\![S]\!]p} \mathsf{unit}$. This type says that `post` is parameterized by a site S and that the data being sent to S carries information about the proposition p. The function type includes in its effect that the site S now *knows* p, as indicated by the notation $[\![S]\!]p$. In particular, if the proposition p is $\mathsf{password}(\mathsf{I}, \mathsf{P})$, then the type of post indicates that S has learned the password as a result of its execution. In this manner, type checking can be used to ensure that a program conforms to restrictions on the flow of knowledge specified in the security protocol.

One challenge with respect to this aspect of the language design is sharing names of objects between the policy and program interface. Another challenge is tracking the correspondence between propositions (like p in the example above) and the data in the type system. A third challenge is dealing with implicit information flows—some care must be taken to ensure that information about the calling context of the function doesn't leak inappropriately.

Contributions To summarize: (1) We will develop a Proof-Carrying Authorization API for JavaScript and the accompanying run-time infrastructure necessary to implement the API. (2) We will develop a programming language whose information-flow policies are specified in a way compatible with the authorization logic, following the ideas sketched above. (3) We will develop techniques to reconcile the *static* parts of the information-flow policy specified in the program text itself with the *dynamic* authorization policies that are enforced by the run-time system. The connection between static and dynamic policies will take the form of programming language constructs (like the `dynamicallyAuthorized` operation described above) whose static typing rules reflect the results of dynamic checks. In particular, such operations are likely to return proofs witnessing the authority so obtained. (4) We will develop type- and proof-inference algorithms for this language, to reduce the amount of explicit proofs that programmers need to deal with directly. Section 2.4 describes our implementation plans.

Related work Language-based security has a long history [70]; here we survey only the most closely related work. A good deal of recent research tackles the problem of making language-based information flow enforcement more practical [69, 18, 48, 71]. Our own work on declassification [82, 81, 47, 56] and dynamic security policies [77, 37, 78] will certainly influence the design of the proposed language; the novel contributions proposed here involve the use of mechanical theorem provers (see Section 2.3 below) and the use of a much more general policy logic. With respect to combining programming language type systems with authorization logic, Chaudhuri and Abadi's recent work on combining file-system access controls with typechecking [17] is similar in spirit to our combination of static and dynamic enforcement. Abadi has also studied access control in the dependency core calculus [4], which is closely related to our own authorization logic [29]. However, his work considers only the "affirmation" components of the logic we have proposed—the novel contributions here are linearity and the "may know" modality. Two prominent implementations of languages with support for information-flow policies are Jif [55, 54] (based on Java), and FlowCaml [66, 73] (based on OCaml). Both languages support fairly rich label models, but neither is as general as the logical approach proposed here. Of the two, Jif supports some limited forms of dynamic policy queries through its `actsFor` tests. We are not aware of any implementations that carry our idea of manifest security through all levels of the system, though several other projects have applied combinations of programming language tools or logical specifications to achieve high-assurance implementations of secure systems. Chothia, Duggan, and Vitek's KDLM system [19] provides a programming language that synthesizes cryptographic protocols based on the policy specified in the language. Fournet, Gordon, and Maffeis [27] designed and implemented a variant of the spi-calculus [1] with a type system capable of enforcing high-level authorization policies described as simple logic programs. Their language extends earlier work by Gordon and Jeffrey [33, 32, 34].

2.3 Reasoning About Policies and Languages

All the proposed work discussed thus far relies on the soundness of a logic or language. Compliance with a formal security policy is meaningful only to the extent to which the policy actually means what it seems to mean. If the basic logic is inconsistent, or if the policy is defective in some manner (either by saying too much, thereby preventing useful work, or too little, thereby permitting an attack), the fact that a program complies with the policy becomes useless. Therefore, it is necessary to *prove* that the logic is consistent, and that a policy of interest is coherent. Similarly, that a program passes the type checker for a security-typed programming language is meaningful only if the type system is sound, and therefore it is necessary to prove the soundness of our languages' type systems.

Thus, an essential component of the work will be the formal analysis of the metatheory of the logic, policies, and programming languages we develop. For the logic, the principal metatheoretic results we will obtain are the cut elimination property, and, largely as a corollary, the logic's soundness relative to a formal semantics. Similarly, for the programming language(s), the principal metatheoretic results will be soundness of the type system and non-interference. For policies, the metatheory will consist of the analysis of policies (using the logic's cut elimination property) to show exactly what they entail.

Experience has shown that formal metatheory for full-scale, fully featured languages (or logics) are infeasible without machine assistance. This difficulty arises for two related reasons. First, distinct language features that are usually studied in isolation rarely turn out to be truly orthogonal, resulting in pervasive complexity. For example, even so simple a feature as a mutable store interacts in some manner with nearly every construct in the language. Second, the sheer number of cases for each lemma is unmanageable for a full-scale language. Even if a hero theorist were able to develop such a soundness proof, and do it correctly, it would be vanishingly unlikely that anyone else would ever read and verify the entire proof. It is therefore our aim that our proofs of the metatheoretic properties of logics, policies, and languages be done in machine-checkable form to the greatest extent possible.

Moreover, there is an additional advantage that we can realize by proving the soundness of our security-typed languages in machine-checkable form. By doing so, we can allow the automatic integration of new programming languages into our architecture. If a software developer is unsatisfied with the standard programming languages available, she can invent her own and give a machine-checkable proof of its soundness. Our architecture can check that proof automatically, and (assuming it passes) add the new language to its set of acceptable languages.

To carry out this mechanized metatheory, we propose to use an automatic proof assistant to formalize languages, policies, and the underlying logic and to verify their security properties to the greatest extent possible. The tool that seems most promising for this purpose is Twelf [64], which has already been used in a number of similar, large-scale experiments such as Foundational Proof-Carrying Code [51, 9, 8], Typed Assembly Language [21, 22], and Standard ML [43]. We propose to investigate how Twelf can be used to verify the meta-theory of our languages, which may in some cases require us to develop new, more syntactic and scalable proof methods or to enrich the framework, for example, to include linearity.

Contributions There are four essential components to making the meta-properties of our logics and languages manifest: (1) We will formalize our logics and languages, including their rules of proof and computations. Such formalization, besides forming the basis of our mechanical development, will also serve to precisely specify the semantics our logics, languages and security model. The necessary science for such formalizations is fairly well-understood for languages with sequential semantics and hygienic binding structure. (2) We will develop and mechanically validate proofs that the semantics of our languages satisfy the type soundness property. As discussed below, the technology to do so in general is now entering the state of the art. In our setting there exist several new challenges that will require new contributions. The first is scale; mechanical verification has yet to be applied in to any language as large as proposed here. (The largest existing application, to Standard ML [43], involves none of the complexity of security-type languages.) Beyond addressing scale, we must develop proof techniques that are effective for linear and other substructural type systems, and also ones appropriate to any new representation techniques we develop. We envision not only new strategies for using existing tools, but also new tools, such as a proof assistant based on CLF [80]. (3) We will develop and mechanically validate proofs about the consistency of our logics, such as cut elimination and strengthening. In this area we will face challenges similar to those for type soundness. (4) We will develop and mechanically validate proofs that our languages' type system imply security policies about distributed information, such as noninterference. This area poses special challenges; part of the proposed research will be to determine the best way to set up such proofs. There are two established techniques for showing noninterference properties. The first, based on *logical relations* [74, 5], does not scale well to full programming languages, which include features such as mutable state. Furthermore, proofs using this technique cannot be naturally represented in some proof assistants. The second, based on a non-standard operational semantics [67, 57], is more compatible with programming language features, but leads to a large number of cases and syntactic complications. At present, this second method seems to be the more promising approach, but we will need to investigate automation techniques to make it feasible.

Related Work There is a growing body of literature on using mechanized theorem provers for verifying the metatheory of programming languages. The principal tools in active use for this purpose are Coq [38], Isabelle/HOL [59], Twelf [64], HOL [35], ACL2 [41], and PVS [61]. All of these tools, with the exception of Twelf, are fully general theorem proving systems capable of formalizing a broad body of mathematics, including the mathematics needed for programming language metatheory. Twelf, by contrast, is specifically tailored to the definition and mechanized analysis of formal systems, including logical systems and programming languages. Considerable successes have been achieved using both approaches. As a recent benchmark of the capabilities of these systems it is useful to mention the solutions to the POPLmark Challenge posed by Pierce, Weirich, Zdancewic, and their collaborators. These solutions are all available from the POPLmark web site [65], and include submissions and by Harper, Crary, and Ashley-Rollman, using Twelf; by Leroy and Vouillon, using Coq; and by Berghofer, using Isabelle/HOL. Beyond POPLmark, researchers have achieved substantial successes in the verification of safety and security properties of programming languages, notably Nipkow, *et al.*'s work on Java and Jinja [58, 42], and Lee, Crary, and Harper's work on Standard ML [43]. Appel's Foundational PCC Project at Princeton [51, 9, 8] has achieved substantial progress on developing the metatheory of low-level assembly languages suitable as target languages for certifying compilers, as has Crary's work on the TALT framework developed at Carnegie Mellon [21]. More recently, Leroy has mechanically verified a back-end for a C compiler [44]. Noninterference results for security type systems and analyses have also been mechanically verified before: David Naumann verified a secure information flow analyzer for a fragment of the Java language [57] and Jacobs, Pieters and Warnier [39] showed noninterference for a simple imperative language in the PVS theorem prover. Also Strecker showed noninterference for MicroJava in Isabelle/HOL [75]. These results differ from our project in terms of scale—we intend to formalize the properties of a much larger programming language with a much richer policy logic.

2.4 Evaluation and Dissemination

Our aim in this project is to develop all necessary mathematical and logical foundations and supporting implementation techniques to create manifestly secure extension platforms in a range of different settings. To demonstrate success, we must show several things. First, the security logic and associated programming language must be internally consistent and have good meta-theoretic properties (such as cut elimination and focusing for the logic, type-soundness and non-interference for the language). Second, the programming language must be rich enough to build a range of applications and easy enough to use in practice that such applications can be built by ordinary developers, not just trained logicians or programming language researchers. Third, the security policy language must be expressive enough to capture sensible policies for these applications and easy enough to use for the people that write these policies. (As discussed above, these will not necessarily be end users, but rather developers and expert policy analysts, supported by the policy analysis tools that we will write.) And fourth, there must be no insurmountable engineering problems involved in deploying this technology in realistic settings.

The first of these criteria—good meta-theoretic properties—will be evaluated by constructing mechanically verified proofs in Twelf as described in Section 2.3. For the others, a concrete demonstration is required.

To this end, we will (1) construct a concrete instance—a manifestly secure extension framework for the Firefox browser—as described in Section 2.2; (2) use this framework ourselves to build several demonstration extensions; and (3) train both graduate and undergraduate students from outside the project to use the framework themselves. In addition we will build a toolkit for establishing properties of policies (such as noninterference), which can used independently of the specific extension framework since it refers only to the security logic itself. We will disseminate all these software artifacts under an open-source license.

Firefox should make an excellent testbed for the concepts and technologies of manifest security:it has an active community of developers implementing a wide variety of extensions—there are dozens available, providing functionality ranging from streamlining blog edits to managing web-site passwords [26].

Our first concrete goal is to reimplement several of these extensions. Reimplementing existing extensions gives us both a clear starting point—a well worked-out design serving an established practical need with real users—and a clear metric of success: whether we can build a new implementation with the same functionality as the original and with formally specified and verified security properties. These extensions are fairly complex pieces of code (typically thousands of lines, using hooks into many parts of Firefox's core functionality and user interface) and have quite sophisticated authorization and/or and information-flow requirements. Success in this domain should therefore give us confidence that our techniques can be transferred to other extensible software platforms.

Each of the extensions we build will consist of two parts: a security policy expressed in our logic and a program implementing the extension itself, carrying with it enough type information for our verifier to certify its compliance with the policy. One specific example that we have already discussed is a secure re-implementation of the default *password manager* functionality (see Section 2.1 in particular). A more sophisticated demo in the same domain could be based on the *Password Hasher* extension [62], which generates strong passwords for multiple websites, using a single master key, and automatically supplies them when these sites are visited. The vulnerabilities are much the same as for the ordinary password manager, but the policy will have to be more complex because of the common master key used in the hashing. Another extension that we plan to reimplement is *Deja Click* [24], a "web recorder" and bookmarking utility that can record sequences of actions (following links, filling in forms, etc.) and replay them when requested. This functionality opens the door to many security vulnerabilities: the extension must have access to browsing history and even passwords, and it must be able to send this information to more or less arbitrary sites. A useful security policy would say, for example, that the URL sequences and form inputs generated by the extension must match prior user inputs (to the same sites), and that no data at all should be given to any other sites. We also plan to implement some extensions with completely new functionality such as support for *Digital Rights Management* policies, which intrinsically require linearity and fair atomic exchange.

Next, we must demonstrate that *others* can write (a) security policies and (b) programs in our language. Our primary plan for doing this is by presenting our tools in classes (undergraduate and graduate classes at Penn and CMU and the annual Summer School on Software Security held at the University of Oregon) and getting students to write policies and programs using them. This strategy gives us a good-sized pool of intelligent but relatively unsophisticated (in security policy, fancy type systems, etc.) developers, providing a good test of the technology's ease of use under relatively controlled conditions.

Finally, we will disseminate our software artifacts to the Firefox developer community (and others). A critical aspect of this process will be providing a number of different applications together with their security policies—many more applications are created by modifying others than from scratch, and more languages are learned by reading code than by reading papers!

2.5 Assumptions, Limitations, and Non-Goals

Here, as with any proposed technique for enforcing security policies, it is necessary to make some assumptions about the context in which the system will be deployed. These assumptions help to delimit the scope and provide traction on the issues at the heart of the project. Our work is based on combining standard cryptographic techniques (such as digital signatures) with novel methods in formal logic and type theory to implement a manifestly secure extension architecture. The primary assumptions, limitations, and explicit non-goals are as follows.

First, we assume that the implementation of the extensible platform on which we build the runtime is trustworthy. We do not propose to verify the correctness of the run-time system, and, for the purposes of this research, we propose to trust the compiler or interpreter that implements the security-typed programming language. (In future work we hope to permit construction of certifying compilers so as to eliminate this assumption, but we consider this direction to be outside the scope of the present work.)

Second, we employ standard cryptographic techniques to ensure confidential and authenticated communication between hosts in the network. For the purposes of mechanical verification, we make the standard Dolev-Yao assumptions [25] and treat encryption operations as perfect. We trust the correctness of the implementation of standard cryptographic techniques such as public-key cryptography and cryptographic hashing and signing methods.

Third, for the sake of practicality and tractability, some parts of the implementation behavior will not be modeled by the security logic. For example, low-level details about caching and timing effects will be omitted, and we will assume that it is intractable for the attacker to perform complete network traffic analysis. Consequently, information-flow security may be circumvented through attacks at a lower level of abstraction than provided by the trusted extensible platform. Existing work on preventing low-level timing [7] and network traffic analysis attacks could in principle be applied in our context.

References

[1] Abadi and Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148, 1999.

[2] Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, pages 228–233, Ottawa, Canada, June 2003. IEEE Computer Society Press.

[3] Martín Abadi. Access control in a core calculus of dependency. In J.H. Reppy and J.L. Lawall, editors, *Proceedings of the 11th International Conference on Functional Programming (ICFP'06)*, pages 263–273, Portland, Oregon, September 2006. ACM Press.

[4] Martín Abadi. Access control in a core calculus of dependency. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006.

[5] 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.

[6] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[7] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[8] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86, Copenhagen, Denmark, July 2002.

[9] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, June 2001.

[10] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[11] Adam Barth and John C. Mitchell. Managing digital rights using linear logic. In *Proceedings of the 21st Symposium on Logic In Computer Science (LICS'06)*, pages 127–136, Seattle, Washington, August 2006. IEEE Computer Society Press.

[12] Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization.* PhD thesis, Princeton University, November 2003.

[13] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, Singapore, September 2005. Springer Verlag LNCS 3650.

[14] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In V. Paxon and M. Waidner, editors, *Proceedings of the 2005 Symposium on Security and Privacy (S&P'05)*, pages 81–95, Oakland, California, May 2005. IEEE Computer Society Press.

[15] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[16] Kevin D. Bowers, Lujo Bauer, Deepak Garg, Frank Pfenning, and Michael K. Reiter. Consumable credentials in logic-based access-control systems. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, California, February 2007. Internet Society. To appear. Preliminary version available as Technical Report CMU-CYLAB-06-002, Carnegie Mellon University, February 2006.

[17] Avik Chaudhuri and Martín Abadi. "secrecy by typing and file-access control". In *Proc. of the 19th IEEE Computer Security Foundations Workshop*, 2006.

[18] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[19] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, 2003.

[20] Jason Crampton, George Loizou, and Greg O' Shea. A logic of access control. *The Computer Journal*, 44(1):137–149, 2001.

[21] Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisiana, January 2003.

[22] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *Nineteenth International Conference on Automated Deduction*, Miami, Florida, 2003. Extended version published as CMU technical report CMU-CS-03-108.

[23] Rocco de Nicola, Gian Luigi Ferrari, and R. Pugliese. klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 1998.

[24] `https://addons.mozilla.org/firefox/3262/`.

[25] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[26] `https://addons.mozilla.org/firefox/extensions/`.

[27] Fournet, Gordon, and Maffeis. A type discipline for authorization policies. In *ESOP: 14th European Symposium on Programming*, 2005.

[28] Deepak Garg, Lujo Bauer, Kevin Bowers, Frank Pfenning, and Michael Reiter. A linear logic of affirmation and knowledge. In D. Gollman, J. Meier, and A. Sabelfeld, editors, *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*, pages 297–312, Hamburg, Germany, September 2006. Springer LNCS 4189.

[29] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In J. Guttman, editor, *Proceedings of the 19th Computer Security Foundations Workshop (CSFW'06)*, pages 283–293, Venice, Italy, July 2006. IEEE Computer Society Press.

[30] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[31] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[32] Gordon and Jeffrey. Typing correspondence assertions for communication protocols. *TCS: Theoretical Computer Science*, 300, 2003.

[33] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *csfw02*, pages 77–91, 2002.

[34] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

[35] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[36] J. Alex Halderman and Edward W. Felten. Lessons from the Sony DRM episode. In *Proc. 15th USENIX Security Symposium (USENIX Security 2006)*, 2006.

[37] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

[38] INRIA. *The Coq Proof Assistant*, reference manual 8.0 edition, January 2005.

[39] B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *Workshop on Issues in the Theory of Security (WITS'05)*, 2005.

[40] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 31. IEEE Computer Society, 1997.

[41] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[42] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[43] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Thirty-Fourth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, Nice, France, January 2007. To appear.

[44] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[45] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[46] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 58–73. Springer-Verlag, 2003.

[47] Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 158–170, January 2005.

[48] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, volume 3302 of *LNCS*, pages 129–145. Springer Verlag, 2004.

[49] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[50] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society Press, May 1994.

[51] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *17th International Conference on Automated Deduction (CADE-17)*. Springer-Verlag (Lecture Notes in Artificial Intelligence), June 2000.

[52] Microsoft Corporation. Microsoft Common Language Runtime System. `http://msdn2.microsoft.com/en-us/netframework/aa663296.aspx`.

[53] Mozilla Foundation. About Javascript. `http://developer.mozilla.org/en/docs/About_JavaScript`.

[54] Andrew C. Myers, , Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[55] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, January 1999. Ph.D. thesis.

[56] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006. To appear.

[57] David Naumann. Verifying a secure information flow analyzer. In *Theorem Proving in Higher Order Logics (TPHOLS)*, 2005.

[58] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a java-like language. In *Proceedings of the Marktoberdorf Summer School*. NATO Science Series, 2003.

[59] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer Verlag LNCS 2283, 2002.

[60] Kevin O'Neill and Joseph Y. Halpern. Secrecy in multiagent systems. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, pages 32–46, 2002.

[61] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[62] `https://addons.mozilla.org/firefox/3282`.

[63] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[64] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[65] The POPLmark Challenge. `http://fling-l.seas.upenn.edu/∼plclub/cgi-bin/poplmark`.

[66] 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.

[67] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 5th International Conference on Functional Programming*, pages 46–57, Montreal, Canada, 2000. ACM Press.

[68] Harald Rueß and Natarajan Shankar. Introducing Cyberlogic. In *Proceedings of the 3rd Annual High Confidence Software and Systems Conference*, Baltimore, Maryland, April 2003.

[69] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, LNCS 2477, pages 376–394, Madrid, Spain, September 2002. Springer-Verlag.

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

[71] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. of the 18th IEEE Computer Security Foundations Workshop*, 2005.

[72] Second Life. `http://secondlife.com`.

[73] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[74] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.

[75] Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[76] Sun Microsystems. *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995. Available at ftp://ftp.javasoft.com/docs/vmspec.ps.zip.

[77] Stephen Tse and Steve Zdancewic. Designing a Security-typed Language with Certificate-based Declassification. In *Proc. of the 14th European Symposium on Programming*, 2005.

[78] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *Transactions on Programming Languages and Systems*, 2006. To appear.

[79] Us-cert vulnerability notes database. `http://www.kb.cert.org/vuls`.

[80] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[81] 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.

[82] 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.

[83] Philip Zimmerman. *PGP(tm) Users's Guide*, October 1994.