Manifest Security for Distributed Information

Karl Crary Robert Harper Frank Pfenning
Carnegie Mellon University

Benjamin C. Pierce Stephanie Weirich Stephan Zdancewic
University of Pennsylvania

March 6, 2006

Project Summary

What is the best way to build programs that compute with data sources controlled by multiple principals, while ensuring compliance with the security policies of the principals involved? The objective of this project is to devise methods for building *manifestly secure* applications for an *information grid* consisting of multiple data sources controlled by multiple principals. This is achieved by using techniques from mathematical logic, programming language semantics, and mechanized reasoning to ensure security of application code, while permitting convenient expression of complex computations with data sources on the information grid. The project will design and implement a programming language whose type system ensures compliance with security policies through the use of proofs in a formal logic of authorization during both the static and dynamic phases of processing. The project will use automated reasoning tools such as theorem provers and logical frameworks to prove formally and rigorously the security properties of the programming language. As a result, every application written in the language enjoys the guarantees afforded by the language as a whole.

The **intellectual merit** of the project consists of scientific and engineering techniques for building practical programs for computing with multiple data sources that are manifestly secure. Manifest security means that the trust relationships, access control and information flow policies, and proofs of compliance with these policies are made manifest in the framework through the use of formal logical methods for specifying and verifying them. These properties will be formally verified against precise specifications written in a novel logic of authorization and information flow using mechanized theorem provers and logical frameworks so that there is a direct link between the theoretical analysis and the executable code. This ensures that running applications are manifestly in compliance with the security policies of the principals on the information grid to an extent not previously achievable in practical systems. The project will build a secure information grid and associated applications to demonstrate the effectiveness of its approach and provide a means for comparison with competing methods.

The **broader impacts** of the project include the development of fundamental technology to ensure privacy while permitting flexible access to disparate, and independently controlled, data sources. Making security policies themselves, and proofs of application compliance with them, readily available in machine-checkable form is a technical cornerstone for ensuring privacy without unduly limiting the legitimate use of these data sources. The project will also significantly increase collaboration between two major research universities within the Commonwealth of Pennsylvania. 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 undergraduate and graduate level.

Project Description

1 Overview

Managers of information repositories today face a tension between security and accessibility of the information they control. Ideally, a manager wishes to make some set of information readily and conveniently accessible for legitimate uses, while still ensuring that no information is leaked or modified inappropriately or without authorization. However, present-day solutions provide good facilities only for one side or the other. On the security side, it is fairly well understood how to prevent unauthorized leakage or modification of information by limiting data accesses to properly privileged queries. This works well provided the gatekeeping and authentication protocols are sound and correctly implemented. Yet, even when all works correctly, access is typically limited to a pre-specified set of queries. Moreover, since protocols and implementations often are not correct, it makes good sense to protect particularly sensitive information by keeping it off the network entirely. On the accessibility side, there has been considerable progress in the grid computing community on the problem of mobile code, wherein programs are able to move across the network to their data of interest and freely compute with it there [28]. This paradigm allows for considerable flexibility in the exploitation of information, because participants may supply their own code to be run by other sites. However, such projects have typically assumed that all participants trust each other. What security is provided focuses mainly on authentication and simple access control, providing no assurance that the accessed information is used appropriately. Research on certified code [70, 61, 95, 22] has sought to weaken these assumptions of mutual trust. However, the work in certified code has focused on preventing mobile code from attacking its host or bypassing its host's access controls, not on complex reasoning about authorization or controlling the propagation of information. This tension between security and accessibility, together with the added problem of correctness, has led to an unsatisfying state of affairs. Information is either (1) accessible but insecure, (2) believed to be secure but not very accessible, or (3) definitely secure but completely off-line.

We propose to develop the theoretical and engineering basis for a *secure information grid.* In our proposed framework, code will be free to move throughout the grid, but before such code may be executed, it must establish to its host that it complies with the host's policy regarding use and propagation of its data.

Architecture The overall architecture of an information grid is shown in Figure 1. There are several key components to an information grid. First, we have a rich *policy language*, expressive enough to specify both authorization and information-flow policies. Together, these policies regulate the use and propagation of information throughout the system. The policy language is used at all stages of grid software lifecycle: During development, it is used to express constraints on legal program behaviors that can be checked statically. During deployment, policies appearing in code certificates are verified by hosts to ensure compliance with their own local data polices. Finally, during execution of grid software, the policy language provides the vocabulary for authorization checks that are performed at run time to enforce access control. The policy language (described in more detail below) is based on an open-ended authorization logic.

Next, we have a *programming language* and accompanying *certifying compiler* The programming language employs a strong type system that uses the policy language for specification of information-flow and authorization policies. The programming language also provides features for describing the *locality* of data sources and the security policies that govern them. The certifying compiler receives two inputs: the program, including its confidentiality and integrity policy annotations, plus information about the trust relationships between principals and hosts in the information grid. This grid information is again expressed as a collection of statements in the policy language. The compiler produces, in addition to executable output, a mechanically verifiable certificate witnessing the type safety of the resulting object code.

Finally, the *runtime system* provides three services to grid software. First, it checks the certificate

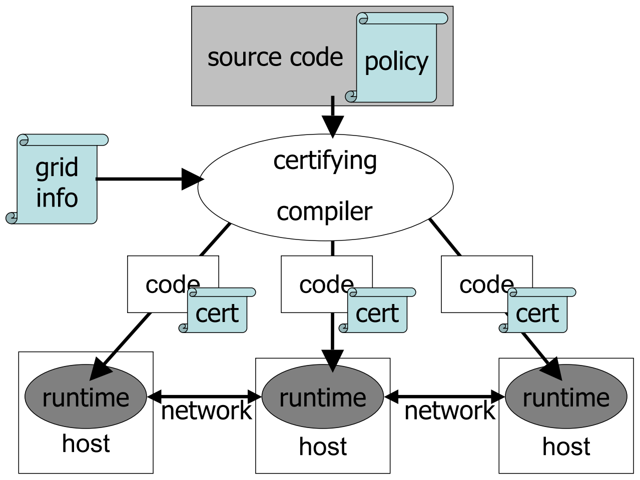

Figure 1: High-level picture of a secure information grid architecture

accompanying any code it is asked to execute—this protects the host against malicious or corrupted code by ruling out potential flaws (like buffer overflows, etc.) and ensures that the code complies with the host's local information-flow policy. This part of the security enforcement occurs before the code is run; the certificate verifier is part of the trusted computing base. Second, the runtime manages digital certificates that represent proof witnesses for the authorization checks the code needs to make as the software is running. And third, the runtime provides secure inter-host communication: when, during the course of execution, a grid program needs to exchange data with or send code to another host in the grid, it does so through the runtime system, which applies appropriate authentication and encryption to ensure that the underlying communication channel is secure.

Manifest Security The unifying theme of the information grid architecture is *manifest security*: all the steps in the chain of reasoning showing that a given program may safely be executed by a given host—including both security policies (including trust relationships among principals and hosts) and the arguments showing that the program obeys them—are made explicit with concrete evidence in the form of machine-checkable proofs.

Manifest security has a number of appealing advantages. Making the type safety and policy compliance of the executable code manifest protects the hosts of the grid from malicious attack. Manifest security also facilitates auditing the behavior of the system. At runtime, explicitly constructed proofs of authorization decisions make it possible to account for the behavior of the system and, when things go wrong (hosts crashing, passwords or keys being compromised, etc.), identify and localize the problem. Statically, making the policies manifest means that it is possible to check them for consistency and to identify assumptions. Mechanical verification of these properties leads to high degree of confidence in the system's correctness.

In practice, the only scalable way to construct evidence of compliance for a large number of grid programs is to begin by verifying, once and for all, the security properties of the programming language in which the programs are written: the *type safety* of the language as a whole then establishes that any given program complies with its stated policy as long as it is well typed. Consequently, the type safety of the source language is of critical importance, and, following the idea of manifest security, it must also be made explicit in a mechanically verifiable way. Thus, a key aspect of the information grid is the mechanical verification of the metatheoretic properties (type safety, noninterference, etc.) of the programming language and accompanying infrastructure.

To summarize, the central idea of the proposed work is making the security of the information grid infrastructure manifest. The soundness of the language is made manifest by a proof carried out explicitly in a formalized metalogic. The fact that a particular executable program accords with the type system of the language is made manifest through the process of typechecking and certifying compilation. Which hosts are allowed to manipulate which data is made manifest by policies that explicitly describe the trust relationships among hosts and principals in the grid. The authorization decisions made by an executing program are made manifest by requiring the system to be able to dynamically produce certificates that correspond to proofs in the authorization logic. And the policies themselves are made manifest as formal objects referred to by these proofs. The upshot of all of this is that we obtain strong assurance that the programming language and compiler are sound, that the security policies used in a system are consistent and consistently enforced, and that any information flows or access control decisions made by the running system comply with the policy.

An Example One of the applications we intend to build to stress-test our infrastructure is an information grid for managing journal paper submissions and reviewing. (We will focus on security and information management issues: implementing a full-blown journal management system with a sophisticated web-browser interface, etc., is beyond the scope of the project.) The application will support multiple journals, with some level of trust but not complete sharing of information among the principals associated with the different journals (for example, each journal will want to keep its reviews and the identity of its reviewers secret). These principals include the editor(s) for each journal, the reviewers, the authors, and the general public. This domain offers many opportunities for formalization of interesting authorization and secrecy policies. (For example, assigned referees can read the paper(s) to which they are assigned, but no others, and only once they are assigned and have agreed; they can submit a report but cannot revise it on their own; they can see other reports [in anonymous form] once a decision is reached, or earlier if the editor authorizes it; accepted papers can be viewed by the general public; etc.) As a more interesting twist, we also allow the journal editors to share information about reviewers: we introduce another principal called the "reviewer clearinghouse," whose job it is to maintain a database of reviewers, with their affiliations, areas of expertise, etc., that can be queried by journal editors. Moreover, the database includes performance information such as average completion times for reviews and "helpfulness scores" assigned by editors of papers for which they have written reviews. Similar features are already implemented in widely deployed journal management systems; their obvious delicacy and potential for abuse make them a perfect case study for an infrastructure like ours, where ownership of information is decentralized and where policies and their verification are both made manifest.

One concrete way of getting relevant information from the journals into the reviewer clearinghouse database is for the implementor of the clearinghouse to write a snippet of code that is distributed to the hosts responsible for each of the journals, to be run whenever a new review is entered (or another event of interest takes place) so that appropriate information can be sent back in the form of a small mobile agent that moves back to the clearinghouse and updates its database. The "security lifecycle" of one of these code snippets is as follows: It is written and compiled on the clearinghouse host, which, during typechecking, verifies its compliance with the policies published by the journal hosts and produces a certificate demonstrating that the object code produced by the compiler is indeed compliant. This certificate is transmitted along with the object code to the journal hosts, which themselves verify the validity of the certificate with respect to the code, before installing it in some internal list of event handlers. Alternatively, the clearinghouse itself might initiate the execution of a similar snippet, sending agents to each of the journals to gather and carry back data in compliance with their policies. Both alternatives involve the notion of a "located computation," which takes place in a specified security context and which can move, consistently with policy, from one such context to another.

Assumptions, Limitations, and Non-Goals 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—the same is true of an information grid. These assumptions help to delimit the scope and provide traction on the issues at the heart of the project. The work we propose here is largely focused on protecting hosts from potentially malicious code and preventing malicious hosts from disrupting global confidentiality and authorization constraints. Our project will also build on existing results and standard techniques (such as the use of public-key cryptography). The primary assumptions, limitations, and explicit non-goals of our project are as follows.

First, we assume an underlying infrastructure suitable for reliably exchanging code and data among the hosts of the grid (for example, as provided by the standard TCP/IP protocols). Hosts participating in an information grid, and, in particular, their runtime systems, may be trusted to varying degrees by the principals involved in the system (these trust relationships are specified by explicit policies); the network is not trusted.

Second, we employ standard cryptographic techniques to ensure confidential and authenticated communication between hosts in the network. For the purposes of our mechanical verification, we make the standard Dolev-Yao assumptions [27] and treat encryption operations as perfect.

Third, for the sake of practicality and tractability, some parts of the implementation behavior won't be modeled. 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, there is the possibility that an attacker able to interact with the information grid at level of abstraction lower than the one we model may be able to circumvent its information-flow policies. Such abstraction-violation attacks are always possible, regardless of where the abstraction boundary is drawn. Existing work on preventing low-level timing [5] and network traffic analysis attacks could in principle be applied in our context, but we will not focus on this here.

Fourth, our work will not address the issues of fault tolerance and reliability that are typically addressed by replication and the use of consensus protocols (although it is likely that those techniques could be fruitfully applied in an information grid). Our proposed work also does not address denial of service attacks—the focus of this work is on integrity of grid software and the confidentiality of data, not on availability of the system.

This list is certainly incomplete, but we believe that the philosophy of manifest security will itself help to identify and articulate additional assumptions and limitations that may arise.

2 Proposed Work

Our approach to building applications with manifestly secure access to distributed 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.

2.1 Grid Security Policies: A Logic of Authorization and Knowledge

To reason about and enforce properties of our infrastructure and of programs executing on it, we must be able to specify security policies. The first component of our proposed research is thus the development of an appropriate policy language for authorization and information flow. Since our goal is to design and implement a flexible, open-ended architecture, the specification language itself must be both expressive and open-ended. Moreover, we wish to reason formally about properties of security policies in order to avoid unintended consequences of policy decisions. Finally, we would like to be able to verify grid software against these security policies. This section sketches some underlying logical principles for the policy language and some preliminary evidence for the viability of our approach. 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. [4, 2]. However, prior work does not completely satisfy our design criteria—in particular, generality and extensibility is 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 judgements and proof rules [31, 53, 78]. The most basic judgment is that of the truth of a proposition, written as $A\ \mathit{true}$. We furthermore need a judgment of *affirmation*, written $K\ \mathit{affirms}\ A$, expressing a policy of K. For example, $K\ \mathit{affirms}\ \mathsf{may\text{-}access}(L, R)$ is a policy statement by principal K that L may access resource R. This implies the *truth* of $\mathsf{may\text{-}access}(L, R)$ if K also controls the resource R. The final ingredient is the standard notion of hypothetical judgment. We write $\Gamma \Longrightarrow A\ \mathit{true}$ and $\Gamma \Longrightarrow K\ \mathit{affirms}\ A$, where Γ is a collection of assumptions of the form $B\ \mathit{true}$ or $K\ \mathit{affirms}\ B$.

We now sketch a sequent calculus for reasoning about authorization. We begin with the so-called judgmental rules which explicate 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 $A\ \mathit{true}$ 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 $K\ \mathit{affirms}\ A$, 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 $K\ \mathit{affirms}\ \mathsf{may\text{-}access}(L, R)$ 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 [9, 10], wherein L supplies such a proof explicitly for validation by a resource monitor implemented in the grid runtime 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 grid architecture will use a logical framework [80] 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 to, for example, 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 throughout the information grid.

Suppose that principal L has been granted access to file R (by presenting a proof of $\mathsf{may\text{-}access}(L, R)$ to the runtime reference monitor). What kind of information flow does this actually entail—i.e., what *knowledge* can various principles now derive?

In order to define a logic of knowledge, we need a new judgment, K *knows* A, where A is a proposition. Clearly, if K knows A, then A should be true. Consequently, any judgement J entailed by A *true* is entailed by K *knows* A:

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

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 knows 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.

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

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

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

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 linear logic [32].

Space permits only the briefest sketch of the resulting logical system. We distinguish between persistent assumptions (including all the ones made so far) and linear assumptions, which must be used exactly once in a proof. Such assumptions can model either consumable, one-time certificates (for example, the permission to submit a review, but only once) or modifiable data (such as a paper that may be revised). Consumable certificates are linear assumptions K *affirms* A, while modifiable data are linear assumptions K *knows* A.

To illustrate some of these ideas in an example, consider the journal reviewing scenario from the overview. The principals are the journals J, the editors E, the reviewers R, authors A. Data include the papers P. We have the following predicates:

$\mathsf{editor}(E, J)$	E is an editor for journal J
$\mathsf{author}(A, P)$	A is author of paper P
$\mathsf{reviewer}(R, P)$	R a reviewer for paper P
$\mathsf{submitted}(P, J)$	paper P has been submitted to journal J
$\mathsf{may\text{-}access}(K, P)$	principal K may access paper P

The authorization policy includes the following: (1) authors may see their own papers; (2) reviewers must agree to a review; (3) reviewers may see assigned papers.

$$\langle J\rangle(\mathsf{author}(A,P) \wedge \mathsf{submitted}(P,J) \supset \mathsf{may\text{-}access}(A,P)) \tag{1}$$

$$\begin{aligned}\langle J\rangle(\mathsf{submitted}(P,J) \wedge \mathsf{editor}(E,J) \wedge (\langle E\rangle\mathsf{reviewer}(R,P)) \wedge (\langle R\rangle\mathsf{reviewer}(R,P))\\ \supset \mathsf{reviewer}(R,P))\end{aligned} \tag{2}$$

$$\langle J\rangle(\mathsf{submitted}(P,J) \wedge \mathsf{reviewer}(R,P) \supset \mathsf{may\text{-}access}(R,P)) \tag{3}$$

The information itself in this example could be distributed, some stored in the journal's database. For example, we write $[\![J]\!]\mathsf{contents}(P, T)$ to mean that the journal knows the contents of paper P is text T. Some sample aspects of an information flow policy: (1) principals may learn the contents of a paper if the journal affirms this; (2) reviewers may know the authors' identity; (3) a clearinghouse may learn the identity of reviewers for given journal, but not which specific papers they review.

$$\mathsf{submitted}(P,J) \wedge (\langle J\rangle \mathsf{may\text{-}access}(K,P)) \supset ([\![J]\!]\mathsf{contents}(P,T)) \supset ([\![K]\!]\mathsf{contents}(P,T)) \tag{1}$$
$$\mathsf{author}(A,P) \wedge \mathsf{submitted}(P,J) \wedge (\langle J\rangle \mathsf{reviewer}(R,P)) \supset [\![R]\!]\mathsf{author}(A,P) \tag{2}$$
$$(\langle J\rangle \mathsf{reviewer}(R,P)) \wedge \mathsf{submitted}(P,J) \supset [\![\mathsf{clearinghouse}]\!](\exists P.\ \mathsf{submitted}(P,J) \wedge \mathsf{reviewer}(R,P)) \tag{3}$$

Permission to update or change both papers and reviews can be added, although the information flow entailed by such actions requires a change of state and therefore a linear logic of knowledge. Similarly, charging for the privilege of viewing accepted papers could be specified using consumable authorities and resources.

A logic of authorization and knowledge as sketched above is so general that it can express policies that are unenforceable or unimplementable. For example, a policy might state that "*If K knows the contents of every file that L owns, then M should be able to learn this fact.*" M and L (and even K) may not have enough information to establish the antecedent, and even a trusted, omniscient party would find it expensive. We therefore propose to identify programming models that go hand-in-hand with classes of policies that they can enforce, employing a combination of logical and cryptographic techniques. A particularly natural class is that of *stratified policies* where information flow may depend on authorization, but not vice versa. Stratification allows us to generate explicit proofs of authorization without directly relying on potentially private knowledge and enables us to use proof-carrying authorization as an enforcement mechanism.

Contributions The major contributions of this thread of research on the logical foundations of authorization and knowledge 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-theoretical 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 (see the next section). (3) We will develop appropriate cryptographic enforcement mechanisms for the linear authorization 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 a chain of formal proof steps ultimately relying on such signed certificates.

Related Work Since Abadi *et al.*'s seminal work [4], there have been numerous proposals for authorization logics [42, 14, 23, 2, 50, 49, 85]. Many of these have different aim and scope from our work in that they are often designed to capture and reason about existing mechanisms, rather than based on purely logical principles. As far as we are aware, these have not 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 [9, 10, 12, 13] with one recent paper taking the first steps at exploring the value of linearity to model consumable credentials [11]. This work, however, has been carried out in the framework of classical higher-order logic 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 [75], 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 [26]. Proof-carrying authorization extends this to a much richer and open-ended policy language at the cost of more complex enforcement mechanisms.

2.2 Enforcing Information-flow Policies

The logic described in the previous section allows information grid applications to specify rich information-flow policies. Unlike authorization policies, which can be enforced by the runtime reference monitor, information-flow policies must be enforced statically [54, 90]—intuitively, this is because information-flows can arise because of what the system did *not* do, and so are a property of the set of all possible system behaviors. It follows that enforcing information-flow policies depends on being able to statically analyze grid software. There is a large body of work developing these language-based techniques (see [87] for an overview), but in the context of our information-grid infrastructure there are a number of new challenges.

One is that existing approaches typically rely on type systems enhanced with rather simple security *classification labels* that restrict information-flows in the program. For example, in the following code fragment, variables `x` and `y` are both integers, but `x` has label L while `y` has label M:

```
int{L} x;  int{M} y;  x = 2 * y;
```

The assignment `x = 2 * y` is permissible only if the label L is at least as restrictive as the label M, written $M \sqsubseteq L$. Intuitively, data with label L must be treated more carefully than data with label M. If the constraint $M \sqsubseteq L$ does not hold, the compiler should reject this program as insecure.

The challenge in our context is to give a logical interpretation to these classification labels that is compatible with both the known program analysis techniques and the policy logic described above. Here, we sketch how this might be accomplished for classification labels based on Myers and Liskov's decentralized label model [67]. Decentralized labels permit *data owners* to specifies sets of *readers* of the data. For example, the label written as $\{K : R_1, \ldots, R_n\}$ indicates that K is an owner of the data and K's policy is that $R_1, \ldots, R_n$ are permitted to read the data. A piece of data might have several owners, each with their own reader constraints. Such labels can be used to implement confidentially policies; data integrity requires additional information to be recorded in the labels.

Logically, such a label can simply be interpreted as a proposition of authorization regarding operations on the variable. For example, the label $\{K : R_1, \ldots, R_n\}$ on value x can be translated to a conjunction of primitive propositions $\mathsf{owns}(K, x) \wedge K \textit{ affirms } (\mathsf{may\text{-}access}(R_1, x) \wedge \ldots \wedge \mathsf{may\text{-}access}(R_n, x))$. In addition, we would have general policies such as the following two:

$$(\forall K.\ \mathsf{owns}(K, x) \supset K \textit{ affirms } \mathsf{may\text{-}access}(L, x)) \supset \mathsf{may\text{-}access}(L, x)$$
$$\forall K.\ \mathsf{owns}(K, x) \supset K \textit{ affirms } \mathsf{may\text{-}access}(K, x)$$

The first policy permits L to access x if all of the owners permit L to access x; the second says that if K owns x then K can access it.

Checking a label condition like $M \sqsubseteq L$ in the example corresponds to proving a logical entailment $M \supset L$: for every principal that can read x, we have to show that they have permission to read y; otherwise an impermissible information flow is created.

The ideas sketched above can be extended to a logical interpretation for richer label systems, such as those for integrity policies that govern write operations to variables. The logical approach also extends naturally to policies that allow delegation relations among principals, as well as robust declassification mechanisms [106, 68], both of which are useful for constructing practical information-flow policies.

A second challenge is to provide programming language constructs that allow running grid code to query, react to, and perhaps even change the dynamic authorization policies present at hosts in the information grid. For example, a policy query might take the following form within the programming language: `if (dynamicallyAuthorized(may-access(L, x))) then P else Q`, where, in the code block P, we can assume that principal L may access resource x, while code in Q cannot make that assumption. What makes this challenging, is that, for such policy queries to be useful, the static analysis of the program must be able to take into account the results of the query. Thus, `dynamicallyAuthorized` cannot simply be a library routine: it interacts with typechecking in nontrivial ways. There is a large design space here that trades off flexibility of the language with feasibility of typechecking. For example, one issue is how first class the query arguments, like $\mathsf{may\text{-}access}(L, x)$, should be. Another question is whether programs should themselves be able to issue new policy statements, and, if so, how to represent and control this dynamic variation of authority.

A third challenge in this area is certified compilation. The target code produced by the compiler should include proof certificates that establish the compliance of the code with respect to the information-flow policy. However, showing that the desired noninterference properties are preserved by compilation is a nontrivial undertaking. One problem that arises is the need to deal with implicit information flows that are caused by control-flow constructs in the source language. Additional problems arise due to potential aliasing of pointer values, so data structures that involve references must also be constrained so that all aliases of a given pointer agree with respect to their security labels. A sound source-language type system for enforcing information-flow policies must account for all of these cases, and such techniques have been well established in the literature [101, 66, 82, 83, 87]. The problem is that compilation may make it harder for a static analysis to see the control-flow structure of the resulting code. It is therefore necessary to make explicit in the target language sufficient information about the source-language control flow so that the same information-flow properties can be established in both levels. Our team has extensive experience with certified compilation [58, 59, 100], including some preliminary results about certified compilation of information-flow languages [107]; here we propose to go further and build a practical implementation of such a compiler.

Contributions We plan to pursue the following research directions in the context of enforcing information-flow policies: (1) 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. This will require clarifying the connections between language-based and logical formulations of noninterference properties. (2) 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 runtime 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. (3) We will build a certifying compiler for this programming language that generates executable code targeted to the grid infrastructure runtime. The main difficulty here is ensuring that high-level information-flow policies are adequately captured at the lower level of abstraction available for object code.

Related work Language-based security has a long history [87]; here we survey only the most closely related work (not already discussed above). Two prominent implementations of languages with support for information-flow policies are Jif [66, 65] (based on Java), and FlowCaml [83, 91] (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. Neither of them use certifying compilation.

Certifying compilation for information-flow languages has been studied in several contexts. Kobayashi and Shirane [47], Medel et al. [55] and Yu and Islam [103] have proposed variants of typed assembly language with support for simple information-flow policies. Both of these works use techniques close to those developed by Zdancewic and Myers' [107] in which linear types describe low-level control flow properties. To our knowledge, none of these approaches have been implemented for a realistic source language.

There has been much recent research related to making information-flow policy languages more practical [86, 21, 52, 88]. Our own work on declassification [106, 105, 51, 68] and dynamic security policies [96, 39, 97] will certainly influence the design of the proposed language; the novel contributions proposed here involve the use of mechanical theorem provers (see the metatheory discussion below) and the application of certifying compilation techniques.

With respect to incorporating logics for security within type systems, Fournet, Gordon, and Maffeis [29] 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 [34, 33, 35].

2.3 Languages for Located Computing

The fundamental feature of an information grid is that it comprises many data sources or repositories distributed across multiple administrative domains. Thus, data sources are inherently *located* by virtue of being controlled by unrelated principals, each with their own security and privacy policies. It is important to note that the concepts of *administrative* locality and *physical* locality are independent notions. For example, the data governed by a principal might be physically distributed across many sites to ensure high availability and reliability, yet would constitute a single locale from the point of view of security policy. Conversely, a single physical repository of data might well be divided into many administrative domains controlled by separate principals. For example, a journal database might be logically divided into separate administrative locales for submitted and published papers, even though they might all reside at the same physical site. To avoid confusion, we will use the term *locale* to refer to an administrative domain, and the term *site* for a physical location. (Although we focus our attention on administrative locality in this project, it also seems possible to model some aspects of physical locality, such as the untrusted nature of communication channels between sites, by introducing notional administrative locales with, say, particularly weak security policies.)

An adequate programming model for an information grid must take account of the existence of disparate locales in order to ensure compliance with the security policies imposed by the various principals on the information grid. We use the term *located computing* for such a model. We propose to build a language for located computing based on an interpretation of modal logic that accounts for administrative locality [16, 57, 43, 64, 63]. Under this interpretation, the modal logic notion of possible worlds correspond to locales, and the typing relation is relativized to these locales. This expresses the notion of a computation executing in the security context of a locale.

A key concept in modal logic is accessibility between worlds, written $\omega \leq \omega'$, which governs the movement of information. In an insecure grid, accessibility might refer simply to network connectivity, but in the context of manifest security, it must be constrained to comply with information flow policies. For example, we might say that $\omega \leq \omega'$ exactly when ω *affirms* $\mathsf{may\text{-}flow}(\omega, \omega')$, indicating that each world controls information flowing from it. When appropriate, explicit proofs for authorization of information flow can be constructed from the policy at run-time and passed between locales as in proof-carrying authorization. One can further internalize the accessibility judgment as a type, which would then allow programs themselves (rather than just the run-time system) to construct and manipulate evidence of accessibility.

Contributions (1) We will develop a modal logic of security based on an interpretation of worlds as administrative locales (and security categories within them), and accessibility as permitted information flows between such worlds. (2) Based on the modal logic of security, we will devise a practical programming language for located computing. This will involve casting modal features in a form convenient for programming, and also integrating those features with other programming language constructs (*e.g.,* structured data, or I/O). (3) We will implement a certifying compiler for our language to generate code suitable for the information grid. This will implement the protocols necessary for located computing, and, at the same time, generate proofs the logic of authorization and knowledge that its output obeys the security policies of its target locale(s). (4) We will investigate the integration of type systems for information flow (as described in 2.2) with the modal type systems for located computing described here. A particular issue is the extension of information flow to languages with first-class functions or data abstraction, which permit formation of objects carrying "private" data that can be compromised by a malicious host; we hope to address this issue by extending techniques developed in the JIF/Split project [108, 109, 104].

Related Work Our approach to located computing is based on a Curry-Howard (propositions-as-types) interpretation of modal logic. The earliest work of this kind is by Borghuis and Feijs [16], who introduced a language for computing with located information based on modal logic that compiled down to shell scripts that used the `ftp` utility to obtain remote data. Our work is based on Pfenning and Davies' judgmental account of modal logic [79], and the spatial interpretation of possible worlds explored by Moody [57], Jia and Walker [43] and Murphy *et al.* [64, 63, 62]. The last two are based on hybrid logics [20, 17] inspired by Simpson's judgmental formulation of Kripke semantics for modal logic [92]. There is also a considerable body of work on modelling and specifying mobile computation in a process calculus setting. This includes especially Cardelli and Gordon's Mobile Ambients [18, 19], De Nicola *et al.*'s access control systems for Klaim [71, 72], Unypoth and Sewell's Nomadic Pict [98], Hennessy *et al.*'s Distributed Pi Calculus [38] and Schmitt and Stefani's type system [89] for the Join Calculus [30]. Several of these systems employ extrinsic modal logics to state and prove properties of programs, but none is based on a Curry-Howard interpretation of modal logic. Also these formalisms tend to emphasize issues of concurrency, which are suppressed in our setting to the implementation level.

2.4 Verifying the Meta-Theory of Security-Typed Languages

Our approach to building a secure information grid relies on languages whose type systems enforce access control and information flow properties. By verifying the security properties of the type system, we can obtain a corresponding theorem about every type-correct program "for free". This represents a significant savings over approaches based on verifying the security properties of programs written in languages that provide no such guarantees. Experience shows that reasoning about languages is the most practical means of proving properties of individual programs. An additional benefit is that it supports the construction of certifying compilers that transfer source-level typing guarantees to executable object code in the form of machine-checkable safety certificates.

The rigorous verification of security properties of programming languages is therefore central to our work. These properties include *type safety* properties, which imply crucial invariants such as memory safety and control-flow safety, and also ensure compliance with access control and authentication constraints. We will also investigate properties such as *non-interference*, which express compliance with information flow properties.

The theoretical techniques required to carry out these verifications are in hand; the challenge is to do them at scale, for languages of sufficient complexity as to be useful for practical programming problems. Rigorous proofs about full-scale languages are unwieldy, because of the sheer number of cases that must be considered. If we are to be truly rigorous, these cannot just be dismissed by saying "the other cases are analogous" — we must check that they really are!

To make this practical we propose to use automatic proof assistants to formalize languages and to verify fully their security properties. A number of powerful proof assistants have been used to verify the metatheory of programming languages. However, it is not clear which are best-suited for our purposes. The challenges that stem from reasoning about security will also force us to evaluate the adequacy of the tools and possibly devise and implement significant extensions to new libraries.

One promising tool that has been used in a number of similar, large-scale experiments in Foundational Proof-Carrying Code [56, 8, 6] and Typed Assembly Language [24, 25] is Twelf [80]. We propose to investigate if and 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.

Another avenue we intend to explore are tools less specialized to the meta-theory of logic and programming languages and instead designed for general mathematics such as Isabelle/HOL [74] or Coq [40]. In principle, these are clearly strong enough to express, say, a proof of non-interference via logical relations, but there are practical obstacles such as how to develop and reuse a theory of binding or a theory of store. One advance in this direction we may try to exploit is the Isabelle/HOL library for nominal logic [99] to capture freshness conditions.

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. (2) We will develop and mechanically validate proofs that the semantics of our languages satisfy the type soundness property. A challenge we see arising from this aspect of the project is the formalization of linear and other substructural type systems. Current methods of intrinsically representing binding in the formal system do not apply to linear systems. A part of this work will be to investigate new methods for this purpose, either via framework extensions as in CLF [102] or via appropriate libraries. (3) We will develop and mechanically validate proofs about the consistency of our logics, such as cut elimination and strengthening. As above, the substructural nature of our logics can make this process more challenging. We expect approaches and solutions to be uniform across this and the previous task. (4) We will develop and mechanically validate proofs that our languages' type system imply security policies about distributed information, such as noninterference. 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* [93, 3], does not scale well to full programming language, 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 [84, 69], 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 [40], NuPrl [7], HOL [36], Isabelle/HOL [74], ACL2 [45], PVS [77] and Twelf [80]. 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 proposers Pierce, Weirich, Zdancewic, and their collaborators. These solutions are all available from the POPLmark web site [81], and include submissions by Leroy and Vouillon, using Coq; by Berghofer, using Isabelle/HOL; and by Harper, Crary, and Ashley-Rollman, using Twelf.

Others 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 [73, 46]. Appel's Foundational PCC Project at Princeton [56, 8, 6] 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 [24]. More recently, Xavier Leroy has used Coq to verify a back-end for a C compiler [48].

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 [69] and Jacobs, Pieters and Warnier [41] showed noninterference for a simple imperative language in the PVS theorem prover. Also Strecker showed noninterference for MicroJava in Isabelle/HOL [94]. 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.

2.5 Implementation and Evaluation

To evaluate the flexibility of our design, verify the practicality of our approach, and demonstrate a complete application of an information grid, our plans include a significant implementation component. There are three main pieces of the implementation—two general purpose tools used in creating information grids, the *runtime infrastructure* and *certifying compiler*, and an instance of a *specific information-grid*.

The runtime infrastructure is responsible for three important tasks: checking the security certificates accompanying code when it is deployed, managing digital certificates for authorization checks while the software executes, and providing secure inter-host communication. The code distribution and verification components will be based on the ConCert grid architecture[22]. To manage digital certificate queries, we expect to use technology similar to that in existing trust management frameworks such as KeyNote [15] or SD3 [44, 37], adapted to the authorization logic proposed here. To implement the necessary encrypted communication channels we will use existing software packages such as OpenSSH [76].

Our team has a great deal of expertise in building certifying compilers [60, 24, 22]. We will adapt current technology on the certifying compilation of security-typed languages. Although this involves substantial work, we believe that, for the most part, existing techniques will be applicable.

Finally, we will use all this infrastructure to implement a specific information grid: the journal management system sketched in the introduction. This application has comparatively sophisticated authorization and information-flow requirements, which will allow us to evaluate the practicality and effectiveness of our proposed techniques.

An additional degree of evaluation is provided by the proposed work on mechanized metatheory. We propose a substantial amount of language design, and those languages must be proven sound. Typically such soundness proofs are verified by peer review, but our work on mechanized metatheory will allow (and indeed demand) very thorough verification before our designs ever face external review. In short, the metatheory work will aid in evaluation of the language design work.

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, 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.

[4] 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.

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

[6] 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.

[7] S. F. Allen, R. L. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The Nuprl open logical environment. In David McAllester, editor, *Automated Deduction–CADE-17: 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer-Verlag, 2000.

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

[9] 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.

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

[11] Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. Consumable credentials in logic-based access control. Technical Report CMU-CYLAB-06-002, Carnegie Mellon University, February 2006.

[12] 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.

[13] 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.

[14] 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.

[15] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[16] Tijn Borghuis and Loe M. G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000.

[17] T. Braüner and V. de Paiva. Towards constructive hybrid logic (extended abstract). In *Elec. Proc. of Methods for Modalities 3*, Nancy, France, September 2003.

[18] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[19] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Symposium on Principles of Programming Languages*, pages 79–92, 1999.

[20] R. Chadha, D. Macedonio, and V. Sassone. A distributed Kripke semantics. Technical Report 2004:04, University of Sussex, 2004.

[21] 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.

[22] The ConCert project home page. `http://www.concert.cs.cmu.edu`.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] Ian Foster and Carl Kesselman. The Globus toolkit. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 11, pages 259–278. Morgan Kaufmann, San Francisco, 1999.

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

[30] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[31] 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.

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

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

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

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

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

[37] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 316–329, Boston, Massachusetts, January 2000. ACM Press.

[38] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDPi: A language for controlling mobile code. Report 02/2003, Department of Computer Science, University of Sussex, October 2003.

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

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

[41] 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.

[42] 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.

[43] Limin Jia and David Walker. Modal proofs as distributed programs. *13th European Symposium on Programming*, pages 219–223, March 2004.

[44] Trevor Jim. SD3: a trust management system with certificate revocation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

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

[46] 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.

[47] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. In *Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS'02)*, 2002.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[55] Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. A typed assembly language for non-interference. In *ICTCS 2005 Ninth Italian Conference on Theoretical Computer Science Certosa di Pontignano*, volume 3701 of *LNCS*, pages 360–374, Siena, Italy, October 2005.

[56] 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.

[57] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, Oct 2003.

[58] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[59] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[60] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *2nd ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[61] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[62] Tom Murphy VII. Mobile types for mobile code. (Also available as technical report CMU-CS-06-112), January 2006.

[63] Tom Murphy VII, Karl Crary, and Robert Harper. Distributed control flow with classical modal logic. In *Computer Science Logic '05*, pages 51–69, Oxford, UK, August 2005.

[64] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *IEEE Symposium on Logic in Computer Science*, pages 286–297, Turku, Finland, July 2004.

[65] 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.

[66] 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.

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

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

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

[70] George Ciprian Necula. *Compiling With Proofs.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1998. (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–98–154.).

[71] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[72] R. De Nicola and M. Loreti. A modal logic for Klaim. In T. Rus, editor, *Proc. of Algebraic Methodology and Software Technology, 8th Int. Conf. AMAST 2000*, pages 339–354. Springer, 2000.

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

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

[75] 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.

[76] The Open BSD Project. *OpenSSH.* `http://www.openssh.com`.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

[82] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, September 2000.

[83] 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.

[84] 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.

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

[86] 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.

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

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

[89] Alan Schmitt and Jean-Bernard Stefani. The M-calculus: A higher-order distributed process calculus. In *Conference Record of the 30th Symposium on Principles of programmming Languages*, pages 50–61, New Orleans, Louisiana, January 2003. ACM Press.

[90] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[91] 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.

[92] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

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

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

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

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

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

[98] Asis Unypoth and Peter Sewell. Nomadic pict: Correct communication infrastructure for mobile computation. In *Conference Record of the 28th Symposium on Principles of Programming Languages*, pages 116–127, London, England, January 2001. ACM Press.

[99] Christian Urban and Christine Tasson. Nominal techniques in isabelle/hol. In R.Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE-20)*, pages 38–53, Tallinn, Estonia, July 2005. Springer Verlag LNCS 3632.

[100] Joseph C. Vanderwaart and Karl Crary. Automated and certified conformance to responsiveness policies. In *Proc. 2005 Workshop on Types in Language Design and Implementation*, Long Beach, California, January 2005.

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

[102] 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.

[103] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. In Peter Sestoft, editor, *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006, Vienna, Austria*, volume 3924 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[104] Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

[105] 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.

[106] 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.

[107] Steve Zdancewic and Andrew C. Myers. Secure Information Flow via Linear Continuations. *Higher Order and Symbolic Computation*, 15(2/3), 2002.

[108] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure Program Partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.

[109] Lantian Zheng, Stephen Chong, Steve Zdancewic, and Andrew C. Myers. Building Secure Distributed Systems Using Replication and Partitioning. In *IEEE 2003 Symposium on Security and Privacy*. IEEE Computer Society Press, 2003.