Dynamic Updating of Information-Flow Policies

Michael Hicks[*] Stephen Tse[‡] Boniface Hicks[†] Steve Zdancewic[‡]
[*] University of Maryland [†] Pennsylvania State University [‡] University of Pennsylvania

Abstract

Applications that manipulate sensitive information should ensure *end-to-end* security by satisfying two properties: *sound execution* and some form of *noninterference*. By the former, we mean the program should always perform actions in keeping with its current policy, and by the latter we mean that these actions should never cause high-security information to be visible to a low-security observer. Over the last decade, security-typed languages have been developed that exhibit these properties, increasingly improving so as to model important features of real programs. No current security-typed language, however, permits general changes to security policies in use by running programs. This paper presents a simple information flow type system for that allows for dynamic security policy updates while ensuring sound execution and a relaxed form of noninterference we term *noninterference between updates*. We see this work as an important step toward using language-based techniques to ensure end-to-end security for realistic applications.

1 Introduction

Increasingly, personal and business information is being made available via networked infrastructures, so the need to protect the confidentiality of that information is becoming more urgent. A typical approach to enforcing data confidentiality is via access control. Unfortunately, access control only governs the release of information, not its propagation. Once a principal (e.g., a user, process, party, etc.) legally reads some data, he can freely share it, whether purposefully or inadvertently, despite the possible wishes of its owner. Instead, we would prefer applications to enforce *end-to-end* security by governing *information flow*: a principal should not, through error or malice, be permitted to transmit confidential information to an unauthorized party.

An information flow control system typically aims to enforce two properties: *noninterference* and *sound execution*. Given a principal hierarchy that defines the relative security levels of various principals, a program satisfies noninterference when it ensures that high security data is never visible, whether directly or indirectly, to low security observers. A program satisfies sound execution if it does not generate errors at run time. A typical way to satisfy these properties is to use a *security-typed language* [14] wherein the standard types on program variables include annotations to specify which principals are allowed to read. If a program type checks under a principal hierarchy, then, it is guaranteed that the program is noninterfering and sound with respect to the hierarchy. Security-typed languages are appealing because these properties are proven in advance of actual execution.

Typical security-typed languages assume that the principal hierarchy remains fixed during program execution. For long-running programs, such an assumption is unrealistic, as policies often change over time, e.g., to perform revocations [6, 2]. On the other hand, simply allowing the principal hierarchy to change at runtime could violate both the soundness and noninterference properties of running programs.

This paper presents a new security-typed language that allows dynamic updating of information-flow policies, particularly the delegation relations in the principal hierarchy. Section 2 defines a typical security-typed source language $\lambda^{\Pi}_{\leq}$, and then Section 4 defines λ^{Π}_{tag} as an extension of $\lambda^{\Pi}_{\leq}$ with *tags* and the ability to accommodate updates to the principal hierarchy. We prove that $\lambda^{\Pi}_{\leq}$ programs can be compiled to λ^{Π}_{tag} programs automatically, that these programs are sound, and that they respect a flavor of noninterference we dub *noninterference between*

$$\begin{array}{lllllllll}
\mathtt{p} & ::= & \mathtt{X} \mid \mathtt{p},\mathtt{p} & \ell & ::= & \mathtt{p}\!:\mathtt{p} \mid \ell,\ell & \mathcal{E} & ::= & \cdot \mid \mathcal{E}\,\mathtt{m} \mid \mathtt{v}\,\mathcal{E} \mid \mathtt{if}\,\mathcal{E}\,\mathtt{m}\,\mathtt{m} \\
\Pi & ::= & \cdot \mid \Pi, \mathtt{p} \leq \mathtt{p} & \mathtt{u} & ::= & \mathtt{bool}_\ell \mid \mathtt{u} \rightarrow \mathtt{u} & & & \\
\mathtt{m} & ::= & \multicolumn{7}{l}{\mathtt{true}_\ell \mid \mathtt{false}_\ell \mid \mathtt{x} \mid \lambda \mathtt{x}\!:\!\mathtt{u}.\,\mathtt{m} \mid \mathtt{m}\,\mathtt{m} \mid \mathtt{if}\,\mathtt{m}\,\mathtt{m}\,\mathtt{m} \mid \mathtt{if}\ (\mathtt{p} \leq \mathtt{p})\ \mathtt{m}\,\mathtt{m}}
\end{array}$$

Figure 1: Syntax of $\lambda_{\leq}^{\Pi}$: principals $\mathtt{p}$, labels ℓ, permission Π, types $\mathtt{u}$, terms $\mathtt{m}$, and holes $\mathcal{E}$.

updates. To our knowledge, ours is the first system to safely permit general updates to the principal hierarchy, including revocations, in security-typed languages. Our discussion of the meaning of noninterference in the presence of revocation, and the definition of the term *noninterference between updates*, is also new. We believe our approach is an important step to making security-typed languages expressive enough to be used in real systems.

2 A Simple Security-Typed Language, $\lambda_{\leq}^{\Pi}$

To make our discussion of policy updates more concrete, we introduce a calculus $\lambda_{\leq}^{\Pi}$, a formalization of the *decentralized label model* [10] (DLM) based on the simply-typed lambda calculus. We present a discussion on policy updates, their challenges, and our solution to making them sound in the following two sections.

Figure 1 presents the syntax of $\lambda_{\leq}^{\Pi}$. Security policies specify confidentiality policies, defining which principals are allowed to read which data. Policies are specified in two parts. First, types and simple values are annotated with *labels* ℓ that consist of one or more pairs ($\mathtt{p}_1 : \mathtt{p}_2$), where $\mathtt{p}_1$ is the policy owner specifying $\mathtt{p}_2$ as the reader. Principals $\mathtt{p}$ can be either literals $\mathtt{X}$ or *principal sets* ($\mathtt{p}_1, \ldots, \mathtt{p}_n$). A label with multiple pairs, written (ℓ_1, ℓ_2) can be used to specify more restrictive policies: a potential reader must satisfy all of the label restrictions. The second part of a DLM security policy is the principal hierarchy, or a permission context, Π is represented as a list of delegations between principals $\mathtt{p}_1 \leq \mathtt{p}_2$.

Terms $\mathtt{m}$ and types $\mathtt{u}$ are largely standard. We write evaluation as $\Pi \vdash \mathtt{m}_1 \longrightarrow \mathtt{m}_2$, which states that $\mathtt{m}_1$ evaluates in a single step to become $\mathtt{m}_2$ under runtime principal hierarchy Π. The principal hierarchy can be accessed dynamically using the run-time test of principal delegation $\mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{e}_1\ \mathtt{e}_2$ [17]:

$$\frac{\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2}{\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\,\mathtt{m}_2 \longrightarrow \mathtt{m}_1} \qquad \frac{\Pi \not\vdash \mathtt{p}_1 \leq \mathtt{p}_2}{\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\,\mathtt{m}_2 \longrightarrow \mathtt{m}_2}$$

The typing judgment has the form $\Pi; \Gamma \vdash \mathtt{m} : \mathtt{t}$, where Γ tracks the types of bound variables as usual, and permission context Π statically tracks knowledge of the principal hierarchy. Most rules are standard [17], as shown in Figure 2. The judgment $\mathtt{lab}(\mathtt{u}) = \ell$ returns the label of the type:

$$\mathtt{lab}(\mathtt{bool}_\ell) = \ell \qquad \frac{\mathtt{lab}(\mathtt{u}_2) = \ell}{\mathtt{lab}(\mathtt{u}_1 \rightarrow \mathtt{u}_2) = \ell}$$

There are two additional typing rules. The first one type-checks the run-time test of principal delegation. If the principal delegation test succeeds, the first branch can statically assume that $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$ holds inside by adding[1] $\mathtt{p}_1 \leq \mathtt{p}_2$ to the permission context Π.

$$\frac{\Pi, \mathtt{p}_1 \leq \mathtt{p}_2; \Gamma \vdash \mathtt{m}_1 : \mathtt{u} \qquad \Pi; \Gamma \vdash \mathtt{m}_2 : \mathtt{u}}{\Pi; \Gamma \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\,\mathtt{m}_2 : \mathtt{u}} \qquad \frac{\Pi; \Gamma \vdash \mathtt{m} : \mathtt{u}_1 \qquad \Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2}{\Pi; \Gamma \vdash \mathtt{m} : \mathtt{u}_2}$$

[1]The second branch *cannot* assume $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$; hence, there is no addition of constraints to the context for the second branch in the typing rule. Adding negative constraints ($\mathtt{p}_1 \not\leq \mathtt{p}_2$) to the context is unnecessary, because subtyping can be decided with positive constraints.

$$\Pi;\Gamma,\mathtt{x}{:}\mathtt{u} \vdash \mathtt{x} : \mathtt{u} \qquad \frac{\Pi;\Gamma \vdash \mathtt{m}_1 : \mathtt{bool}_\ell \quad \Pi;\Gamma \vdash \mathtt{m}_2 : \mathtt{u} \quad \Pi;\Gamma \vdash \mathtt{m}_3 : \mathtt{u} \quad \mathtt{lab(u)} = \ell}{\Pi;\Gamma \vdash \mathtt{if}\ \mathtt{m}_1\ \mathtt{m}_2\ \mathtt{m}_3 : \mathtt{u}}$$

$$\Pi;\Gamma \vdash \mathtt{true}_\ell : \mathtt{bool}_\ell \qquad \frac{\Pi;\Gamma,\mathtt{x}{:}\mathtt{u}_1 \vdash \mathtt{m} : \mathtt{u}_2}{\Pi;\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{u}_1.\,\mathtt{m} : \mathtt{u}_1 \rightarrow \mathtt{u}_2}$$

$$\Pi;\Gamma \vdash \mathtt{false}_\ell : \mathtt{bool}_\ell \qquad \frac{\Pi;\Gamma \vdash \mathtt{m}_1 : \mathtt{u}_1 \rightarrow \mathtt{u}_2 \quad \Pi;\Gamma \vdash \mathtt{m}_2 : \mathtt{u}_1}{\Pi;\Gamma \vdash \mathtt{m}_1\ \mathtt{m}_2 : \mathtt{u}_2}$$

Figure 2: Typing rules of $\lambda^{\Pi}_{\leq}$: $\Pi;\Gamma \vdash \mathtt{m} : \mathtt{t}$ (under hierarchy Π and context Γ, the term $\mathtt{m}$ has type $\mathtt{t}$).

The second one is the subsumption rule that allows the flexibility of implicitly appealing to principal delegations in the permission context during typing. There exist straightforward and efficient algorithms [7, 17] for context subtyping $\Pi_1 \leq \Pi_2$ (meaning Π_1 is more permissive than Π_2), label subtyping $\Pi \vdash \ell_1 \sqsubseteq \ell_2$ (meaning ℓ_1 is less restrictive than ℓ_2 under Π), principal subtyping $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$ (meaning principal $\mathtt{p}_1$ is delegating to $\mathtt{p}_2$ under Π), and type subtyping $\Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2$ (meaning $\mathtt{u}_1$ is a subtype of $\mathtt{u}_2$ under Π); we leave their formal specification to our companion technical report.

We have proved the desired properties of sound execution and noninterference for $\lambda^{\Pi}_{\leq}$ by a sound translation to a target language, which is to be described in Section 4.

3 The Meaning of Policy Updates

Now we consider the means and the meaning of security policy updates in $\lambda^{\Pi}_{\leq}$ (and similar languages). An information-flow security policy can change in two ways. First, the label on a particular piece of data might be altered, thereby making access to it more restricted or less restricted. The former case is permitted automatically by the subsumption rule: it is always safe to treat a piece of data more restrictively. The latter case is potentially dangerous, as the relabeling might expose sensitive information, so it is typically allowed only by an explicit *declassify* operation. A second way of changing the information-flow policy is to alter the principal hierarchy, which in turn alters the relative ordering between labels. Here again there are two kinds of changes: one can add a (directed) edge between two principals (e.g., the delegation $\mathtt{p}_1 \leq \mathtt{p}_2$), which corresponds to increasing the privileges of an observer. This kind of policy change is like a global form of declassification. One can also remove edges, corresponding to revocation, strengthening the security policy and decreasing the set of permissible information flows.

In this paper, we address the second style of policy update in which the principal hierarchy can change. To determine how and when the hierarchy should be permitted to change, we must consider the impact of policy updates on a program's security properties: sound execution and noninterference.

3.1 Models of updating

It is easy to see that naively allowing arbitrary policy updates could make evaluation *unsound*: the program could act in a way not consistent with its current policy. As an example, consider this simple program (extending the syntax presented earlier with `let` and integers):

$$\mathtt{let\ x : int}_{\mathtt{p}_2:} \quad = \quad (\mathtt{if\ (p_1 \leq p_2)}\ 1_{\mathtt{p}_1:}\ 3_{\mathtt{p}_2:}) \qquad \mathtt{in} \qquad \mathtt{if\ (p_2 \leq p_3)\ x}\ 2_{\mathtt{p}_3:}$$

If we evaluate this program under $\Pi = \mathtt{p}_1 \leq \mathtt{p}_2$, after one step the `if` branch succeeds yielding

$$\mathtt{let}\ \mathtt{x} : \mathtt{int}_{\mathtt{p}_2:} \quad = \quad 1_{\mathtt{p}_1:} \qquad \mathtt{in} \qquad \mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ \mathtt{x}\ 2_{\mathtt{p}_3:} \tag{Ex 1}$$

Now say we wish to change the principal hierarchy to be $\Pi' = \mathtt{p}_2 \leq \mathtt{p}_3$. If we allow this update to occur, then the program's next evaluation step will be unsound. It will allow the data $1_{\mathtt{p}_1:}$ to flow to variable `x`, whose label $\mathtt{p}_2$: is not equal or higher under Π'. Clearly, this update should not be permitted.

Noninterference is more subtle because it is a global property: it ratifies all information flows that might occur during a program's *entire* evaluation relative to a single, fixed principal hierarchy. When the hierarchy can change, this definition no longer makes sense. One alternative is *noninterference between updates*, meaning that when a policy update occurs, the history of how some data received a certain label is forgotten, and the question of noninterference is reconsidered for the current program at the current policy. As motivation for this definition, imagine some data (a file, say) labeled as $\mathtt{p}_1$: under $\Pi'' = \mathtt{p}_1 \leq \mathtt{p}_2,\ \mathtt{p}_1 \leq \mathtt{p}_3$, meaning principals $\mathtt{p}_2$ and $\mathtt{p}_3$ are allowed to read it. If principal $\mathtt{p}_3$ is fired and $\mathtt{p}_4$ is hired, we might like to change the principal hierarchy to be $\mathtt{p}_1 \leq \mathtt{p}_2, \mathtt{p}_1 \leq \mathtt{p}_4$; i.e. to revoke the assertion $\mathtt{p}_1 \leq \mathtt{p}_3$ and add the assertion $\mathtt{p}_1 \leq \mathtt{p}_4$. From the point of view of the file labeled $\mathtt{p}_1$: and the original hierarchy Π'', the changed hierarchy both rejects flows previously allowed ($\mathtt{p}_3$ can no longer read the file) and permits flows not previously admitted ($\mathtt{p}_4$ can read the file, but could not before). Thus, at least some portion of the file's information flow history must be forgotten to permit this intuitively reasonable policy change.

On the other hand, noninterference between updates can permit unintuitive, perhaps unintended flows. For example, say the program (Ex 1) takes an additional step under Π yielding

$$\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ 1_{\mathtt{p}_1:}\ 2_{\mathtt{p}_3:}$$

Under the initial principal hierarchy Π, the program would terminate with result $2_{\mathtt{p}_3:}$. However, if we were to change the hierarchy to $\Pi''' = \mathtt{p}_1 \leq \mathtt{p}_2, \mathtt{p}_2 \leq \mathtt{p}_3$, then the `if` branch would be taken, and we would terminate with result $1_{\mathtt{p}_1:}$. The evaluation is sound and noninterfering under Π''' from the point of the change to termination. However, from the point of view of the entire program evaluation, the observed flow would have been disallowed under Π, and thus violates noninterference when considered relative to Π. Moreover, a program that satisfies noninterference between updates says nothing about the security ramifications of updates themselves. For example, one risk is that if an attacker can observe when a policy update occurs and what it consists of, he may be able to deduce the values of private data in the program.

An ideal security property would both permit policy updates to selectively "forget the past" and also reason about some flows across updates. It is an open question as to what information flow systems should enforce even when policy updates are disallowed—noninterference, though commonly supported, is too restrictive in practice. As a first step, for this paper, we ensure that program execution is sound and respects noninterference between updates, recognizing and expecting that a better property is needed. We plan to investigate stronger, adequately expressive security properties in future work.

3.2 Overview of approach

At first glance, defining principal hierarchy updates for $\lambda^{\Pi}_{\leq}$ that ensure soundness and noninterference between updates may seem straightforward. In particular, we can show noninterference between updates by proving the standard notion of noninterference: the type system is parameterized by a fixed principal hierarchy that is enforced as usual, since it implies that as long as the policy does not change, the program is noninterfering.

Proving soundness would seem equally simple: to update the principal hierarchy Π to Π' while running program `m` requires that we simply type check `m` under Π': if type checking succeeds then we permit the update. While this approach is sound (by definition), it is overly restrictive. Consider our example program again. Say the program evaluates under Π and becomes as in Ex 1. Then say we wish to change the hierarchy to be Π'. This program will not type check since the expression $1_{\mathtt{p}_1:}$ cannot be given type $\mathtt{int}_{\mathtt{p}_2:}$ under Π'. But conceptually it should be legal, because x (which we have substituted for here with $1_{\mathtt{p}_1:}$) should be treated as having type $\mathtt{int}_{\mathtt{p}_2:}$, as defined in the original program. This fact is not revealed, however, in the run-time representation of the current state. That is, an important fact of the past (changing the type of $1_{\mathtt{p}_1:}$ to $\mathtt{int}_{\mathtt{p}_2:}$) has been forgotten.

We can solve this problem by moving away from the view of *subtyping as subset* to the view of *subtyping as coercion* for evaluation. Rather than viewing data $1_{\mathtt{p}_1:}$ as having type $\mathtt{int}_{\mathtt{p}_2:}$ under Π, we say that we can *coerce* $1_{\mathtt{p}_1:}$ to a value that has type $\mathtt{int}_{\mathtt{p}_2:}$; that is, we can coerce it to $1_{\mathtt{p}_2:}$. To do this, we extend $\lambda^{\Pi}_{\leq}$ with *permission tags* that act as coercion functions. In particular, the expression $[\ell \sqsubseteq \ell']1_\ell$ will evaluate to $1_{\ell'}$. With this change, our original program becomes

$$\mathtt{let}\ \mathtt{x} : \mathtt{int}_{\mathtt{p}_2:} \quad = \quad \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ ([\mathtt{p}_1{:} \sqsubseteq\ \mathtt{p}_2{:}]1_{\mathtt{p}_1:})\ 3_{\mathtt{p}_2:} \qquad \mathtt{in} \qquad \mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ ([\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]\mathtt{x})\ 2_{\mathtt{p}_3:}$$

That is, the uses of subsumption are made explicit as tags. Then the program will evaluate under Π to become

$$\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ ([\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]1_{\mathtt{p}_2:})\ 2_{\mathtt{p}_3:}$$

Now we can see that changing to Π' will be legal, as 1 has a label that can be properly typed in the new policy. At the same time, we still prevent illegal updates to the policy. In the more general case that $1_{\mathtt{p}_1:}$ were some expression $\mathtt{m}_{\mathtt{p}_1:}$, it would be unsound for the policy to change until m is a base value. Thus, while m is being evaluated (i.e., in the context of the if expression), it is guarded with the tag $[\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]$. An update that violated this constraint would not be allowed (as desired).

In addition to providing a more flexible coercion semantics, it turns out that permission tags can also lead to a more efficient implementation. In particular, rather than having to type check the entire program body at each proposed change in policy, we only need to look at the tags, which succinctly capture how the current policy is being used. Section 4 presents a dynamic traversal that discovers these tags at update points without having to consider function bodies. We conjecture that with only a little more work, we can adjust the evaluation semantics to keep track of the current "tag context". This would allow us to replace the traversal with a simple check.

3.3 Example

To show how these issues might arise in practice, we conclude this section with an example. Figure 3 shows a class for accessing the records of a company database, written in a Java-like syntax. This class defines two run-time principals mgr, which is a division manager, and div, which represents a division of company employees.[2] Lines 5 through 8 define some utility functions getting query inputs from the system user, processing them, creating summaries, and displaying information to the user. The policies on these methods establish that queries and the resulting processed data are owned by the mgr principal and readable by all principals in the group div, but that the results of auditing a query are only readable by mgr. These policies are explicit in the program: for example, the label on line 5 indicates that the result of get_query is owned by the principal mgr and readable by (principals in) the group div; similarly the audit method takes data readable by div and returns data only readable mgr (owners are implicitly considered to be readers in our model).

The method access_records is parameterized by a principal emp (employee), which is the current user of the database system. Line 15 dynamically checks that emp is a member of the division div, whose data is stored encapsulated in this database object. This line results in a runtime check of the principal hierarchy and succeeds only if div $\leq$ emp is true at the time when the check is made. Assuming that check succeeds, employee queries are received, processed, and displayed to the user until the user quits. In this scenario, the program also audits the

[2]Run-time principals represent principals as run-time entities, and could readily be added to our system [17].

```
01.       class Database {
02.         principal div;     /* division group */
03.         principal mgr;     /* manager for the division */
04.
05.         Query{mgr:div} get_query() {...}
06.         Data{mgr:div}  process_query(Query{mgr:div} q) {...}
07.         Data{mgr:}     audit(Data{mgr:div} d) {...}
08.         void           display(principal p, Data{mgr:p} {...}
09.
10.         void access_records(principal emp) {
11.           Query{mgr:div} query;
12.           Data{mgr:emp}  result;
13.           Data{mgr:}     summary;
14.
15.           if (div < emp) {  /* employee is a member of the division */
16.             while (true) {
17.               query = get_query();
18.               if (query == Quit) break;
19.               result  = process_query(query);
20.               summary = audit(result);
21.               display(emp, result);
22.
23.               if (mgr < emp) {  /* employee is a manager */
24.                 display(emp, summary);
25.               }
26.               ...  /* log audit information */
27.             }
28.           } else { abort(); }
29.         }
30.       }
```

Figure 3: Information-flow in a database system with principal delegations.

employee queries, perhaps to generate some statistics useful for making management decisions. The results of the audit process are readable only by managers (i.e. those principals p for which mgr $\leq$ p). For convenience, if the user of the system *is* a manager, the results of the audit are displayed immediately—the dynamic check on line 23 ensures that only managers receive this sensitive data. Presumably the program would also log the auditing information for later inspection by a manager; in this case, the current user is not able to see that data.

The code makes an important assumption: though it checks div $\leq$ emp only once, it assumes that this relationship holds for the entire execution of the `while` loop. A problem arises if this relationship is revoked while the loop executes, say if the employee is fired or just moved to a different division. In this case, an employee who no longer belonged to a particular division would still have access to its files. Even worse, if the employee were made a manager (i.e., introducing mgr $\leq$ emp into the principal hierarchy) in a new division, he would suddenly have privileges not allowed under either policy—he could read files belonging to his original division. These scenarios reveal how policy changes can violate both sound execution and our intuitive notion of noninterference.

Introducing permission tags solves these problems. In particular, to store the returned value of `process_query` into `result`, the label of the returned value must be coerced from `mgr:div` to `mgr:emp`. This will be witnessed by a coercion [mgr:div $\sqsubseteq$ mgr:emp] on `process_query(query)`, which in turn will prevent the revocation of the

$$\begin{array}{lll} \mathtt{t} & ::= & \mathtt{bool}_\ell \mid \mathtt{t} \to \mathtt{t} \\ \mathtt{v} & ::= & \mathtt{true}_\ell \mid \mathtt{false}_\ell \mid \lambda[\Pi]\mathtt{x}:\mathtt{u}.\,\mathtt{e} \qquad \mathcal{E} \ ::= \ \cdot \mid \mathcal{E}\ \mathtt{e} \mid \mathtt{v}\ \mathcal{E} \mid \mathtt{if}\ \mathcal{E}\ \mathtt{e}\ \mathtt{e} \mid [\ell \sqsubseteq \ell]\mathcal{E} \\ \mathtt{e} & ::= & \mathtt{true}_\ell \mid \mathtt{false}_\ell \mid \mathtt{x} \mid \lambda[\Pi]\mathtt{x}:\mathtt{t}.\,\mathtt{e} \mid \mathtt{e}\ \mathtt{e} \mid \mathtt{if}\ \mathtt{e}\ \mathtt{e}\ \mathtt{e} \mid \mathtt{if}\ (\mathtt{p} \le \mathtt{p})\ \mathtt{e}\ \mathtt{e} \mid [\ell \sqsubseteq \ell]\mathtt{e} \end{array}$$

Figure 4: Syntax of λ^{Π}_{tag}: types $\mathtt{t}$, values $\mathtt{v}$, terms $\mathtt{e}$, holes $\mathcal{E}$.

edge div $\le$ emp from the principal hierarchy.[3]

4 A language with dynamic policy and tagging, λ^{Π}_{tag}

This section formally describes an extension to $\lambda^{\Pi}_{\le}$, called λ^{Π}_{tag}, that permits dynamic updates to the principal hierarchy. As just described, we use permission tags of the form $[\ell_1 \sqsubseteq \ell_2]$ to prevent illegal updates of the principal hierarchy during execution. We prove that λ^{Π}_{tag} enjoys the security properties of sound execution and noninterference described earlier, even as policies change at run-time. Permission tag annotations need not burden the programmer; they can be automatically inserted by the compiler. At the end of this section, we present an automatic translation from the source calculus presented earlier to the target calculus here, based on the standard formulation of *subtyping as coercions*, and prove it sound.

The syntax of λ^{Π}_{tag} is presented in Figure 4, and closely matches the source calculus, $\lambda^{\Pi}_{\le}$ in Section 2, except the addition of permission tags. The typing rules are the same, with one exception: the subsumption rule for subtyping is now eliminated, effectively replaced by the new typing rule for tags:

$$\frac{\Pi \vdash \ell_1 \sqsubseteq \ell_2 \qquad \Pi;\Gamma \vdash \mathtt{e} : \mathtt{bool}_{\ell_1}}{\Pi;\Gamma \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{e} : \mathtt{bool}_{\ell_2}}$$

We maintain the invariant that the current principal hierarchy Π always respects the permission tag $[\ell_1 \sqsubseteq \ell_2]$ around the term $\mathtt{e}$, as shown in the judgment $\Pi \vdash \ell_1 \sqsubseteq \ell_2$. Another invariant, which is enforced during the translation in Section 4.2, is that only boolean values are tagged. This permits the following evaluation rules:

$$\Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{true}_{\ell_1} \longrightarrow \mathtt{true}_{\ell_2} \qquad \Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{false}_{\ell_1} \longrightarrow \mathtt{false}_{\ell_2}$$

Functions are not directly tagged: they contain future computations in the body that, unlike booleans, cannot be coerced to work under different security policies. Our strategy is to extend function terms with the permission context, $\lambda[\Pi']\mathtt{x} : \mathtt{t}.\ \mathtt{e}$, such that the context Π' of the function body can be *summarized* to guard against illegal updates. The typing rules for functions and applications are:

$$\frac{\Pi \le \Pi' \qquad \Pi';\Gamma,\mathtt{x}:\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\Pi;\Gamma \vdash \lambda[\Pi']\mathtt{x}:\mathtt{t}_1.\ \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2} \qquad \frac{\Pi;\Gamma \vdash \mathtt{e}_1 : \mathtt{u}_1 \to \mathtt{u}_2 \qquad \Pi;\Gamma \vdash \mathtt{e}_2 : \mathtt{u}_1}{\Pi;\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{u}_2}$$

Given these tags, we can soundly and efficiently check if a policy update is legal during execution. We introduce *dynamic tag checking* $\Pi \vdash \mathtt{e}$, as shown in Figure 5, for ensuring that principal hierarchy Π is valid with respect to the running program $\mathtt{e}$. Any hierarchy is valid against boolean values. Section 4.1 proves that that the dynamic tag checking soundly approximates the static type checking with respect to the validity of policy updates.

At last, we formalize an update to the principal hierarchy of an evaluating program by defining a top-level evaluation relation. Under hierarchy Π, a program can either take a small evaluation step, or change to use the

[3] In a formal operational semantics, loops are implemented by expanding each iteration of the loop into to a fresh version of the original. Each fresh version would contain this tag, preventing any update that would violate it for the duration of the loop's execution.

$$\frac{\Pi \leq \Pi'}{\Pi \vdash \lambda[\Pi']\mathtt{x}\!:\!\mathtt{t}.\,\mathtt{e}} \qquad \frac{\Pi \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2}{\Pi \vdash \mathtt{e}_1\ \mathtt{e}_2} \qquad \frac{\Pi, \mathtt{p}_1 \leq \mathtt{p}_2 \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2}{\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{e}_1\ \mathtt{e}_2} \qquad \frac{\Pi \vdash \ell_1 \sqsubseteq \ell_2 \qquad \Pi \vdash \mathtt{e}}{\Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{e}}$$

$$\Pi \vdash \mathtt{true}_\ell \qquad \Pi \vdash \mathtt{false}_\ell \qquad \frac{\Pi \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2 \qquad \Pi \vdash \mathtt{e}_3}{\Pi \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3}$$

Figure 5: Tag checking $\Pi \vdash \mathtt{e}$ (determines whether principal hierarchy Π is legal for the running program $\mathtt{e}$).

pending hierarchy Π'. The latter step is only permitted if the new hierarchy is legal with respect to the current program, that is, if dynamic tag checking $\Pi \vdash \mathtt{e}$ succeeds:

$$\frac{\Pi \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{(\Pi; \mathtt{e})|\Pi' \longrightarrow (\Pi; \mathtt{e}')} \qquad \frac{\Pi' \vdash \mathtt{e}}{(\Pi; \mathtt{e})|\Pi' \longrightarrow (\Pi'; \mathtt{e})}$$

Note that dynamic tag checking is meant to approximate an implementation. That is, while our formulation requires a traversal over the active part of the program (i.e., the part without functional terms), this traversal could be avoided by statically gathering the set of tags S, S' that appear in the body $\mathtt{e}_1, \mathtt{e}_2$, respectively, of each $\mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{e}_1\ \mathtt{e}_2$ expression, and then annotating the $\mathtt{if}$ with a *tag constraint* $(\mathtt{p}_1 \leq \mathtt{p}_2 \Rightarrow S) \cup S'$ (similar to conditional types [1]). These tag constraints can be maintained to form a *tag context* at run-time, so that dynamic tag checking merely considers the current tag context, rather than the active part of the program.

4.1 Security theorems

To show that the execution of a program written in our calculus is sound, we prove that any well-typed, closed term runs without any error. To show that the information flow satisfies end-to-end security, we prove that any well-typed low-security term is noninterfering by the high-security data. These *type safety* and the *noninterference* properties are formally stated as follows.

Here $\Downarrow$ is the top-level evaluation for the whole program, ignoring the number of policy updates, while $\longrightarrow^*$ is the transitive-closure of the non-updating evaluations. Therefore, type safety is guaranteed during the evaluation of the whole program, but noninterference is guaranteed between updates (as discussed in Section 3.1).

Theorem 1 (Security of dynamic policy updating)

1. *Type safety during execution: If* $\Pi; \cdot \vdash \mathtt{e} : \mathtt{t}$*, then* $(\Pi, \mathtt{e}) \Downarrow (\Pi', \mathtt{v})$.

2. *Noninterference between updates: If (1)* $\Pi; \mathtt{x} : \mathtt{bool}_{\ell_1} \vdash \mathtt{e} : \mathtt{bool}_{\ell_2}$*, and (2)* $\Pi; \cdot \vdash \mathtt{v}_1 : \mathtt{bool}_{\ell_1}$*, and (3)* $\Pi; \cdot \vdash \mathtt{v}_2 : \mathtt{bool}_{\ell_1}$*, and (4)* $\Pi \vdash \ell_1 \not\sqsubseteq \ell_2$*, then* $\Pi \vdash \mathtt{e}\{\mathtt{v}_1/\mathtt{x}\} \longrightarrow^* \mathtt{v}$ *iff* $\Pi \vdash \mathtt{e}\{\mathtt{v}_2/\mathtt{x}\} \longrightarrow^* \mathtt{v}$.

The proof for *type-safety* uses the standard technique of combining the progress and the preservation of a well-typed term. Some important lemmas for showing the soundness of tagging and tag checking are below. The first lemma states a well-typed *value* can also be well-typed under the empty principal hierarchy $\Pi = \cdot$, which is critical in the substitution lemma. The second states that the evaluation rule $\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\ \mathtt{m}_2 \longrightarrow \mathtt{m}_1$ in Section 2 is type-preserving. The last lemma below shows that dynamic checking $\Pi \vdash \mathtt{e}$ is a sound approximation of static type checking $\Pi; \Gamma \vdash \mathtt{e} : \mathtt{t}$.

$$\left[\!\!\left[\frac{\Pi;\Gamma,\mathtt{x}\!:\!\mathtt{u}_1 \vdash \mathtt{m} : \mathtt{u}_2}{\Pi;\Gamma \vdash \lambda\mathtt{x}\!:\!\mathtt{u}_1.\,\mathtt{m} : \mathtt{u}_1 \to \mathtt{u}_2}\right]\!\!\right] \quad = \quad \lambda[\Pi]\mathtt{x}\!:\![\![\mathtt{u}_1]\!].\ [\![\Pi;\Gamma,\mathtt{x}\!:\!\mathtt{u}_1 \vdash \mathtt{m} : \mathtt{u}_2]\!]$$

$$\left[\!\!\left[\frac{\Pi;\Gamma \vdash \mathtt{m} : \mathtt{u}_1 \qquad \Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2}{\Pi;\Gamma \vdash \mathtt{m} : \mathtt{u}_2}\right]\!\!\right] \quad = \quad [\![\Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2]\!]\ [\![\Pi;\Gamma \vdash \mathtt{m} : \mathtt{u}_1]\!]$$

$$[\![\Pi \vdash \mathtt{bool}_{\ell_1} \preceq \mathtt{bool}_{\ell_2}]\!] \quad = \quad \lambda[\Pi]\mathtt{x}\!:\!\mathtt{bool}_{\ell_1}.\ [\ell_1 \sqsubseteq \ell_2]\ \mathtt{x} \qquad \text{(fresh } \mathtt{x})$$

$$\left[\!\!\left[\frac{\Pi \vdash \mathtt{u}_3 \preceq \mathtt{u}_1 \qquad \Pi \vdash \mathtt{u}_2 \preceq \mathtt{u}_4}{\Pi \vdash \mathtt{u}_1 \to \mathtt{u}_2 \preceq \mathtt{u}_3 \to \mathtt{u}_4}\right]\!\!\right] \quad = \quad \begin{array}{l}\lambda[\Pi]\mathtt{x}_1 : [\![\mathtt{u}_1]\!] \to [\![\mathtt{u}_2]\!].\ \lambda[\Pi]\mathtt{x}_2 : [\![\mathtt{u}_3]\!]. \\ \quad [\![\Pi \vdash \mathtt{u}_2 \preceq \mathtt{u}_4]\!]\ (\mathtt{x}_1\ ([\![\Pi \vdash \mathtt{u}_3 \preceq \mathtt{u}_1]\!]\ \mathtt{x}_2))\end{array} \qquad \text{(fresh } \mathtt{x}_1, \mathtt{x}_2)$$

Figure 6: Translating principal delegations to permission taggings.

Lemma 2 (Soundness of dynamic tag checking)

1. *If* $\Pi;\Gamma \vdash \mathtt{v} : \mathtt{t}$, *then* $\cdot;\Gamma \vdash \mathtt{v} : \mathtt{t}$.
2. *If* $\Pi, \mathtt{p}_1 \le \mathtt{p}_2;\Gamma \vdash \mathtt{e} : \mathtt{t}$ *and* $\Pi \vdash \mathtt{p}_1 \le \mathtt{p}_2$, *then* $\Pi;\Gamma \vdash \mathtt{e} : \mathtt{t}$.
3. *If* $\Pi;\cdot \vdash \mathtt{e} : \mathtt{t}$, *then* $\Pi \vdash \mathtt{e}$. *Moreover, if* $\Pi;\cdot \vdash \mathtt{e} : \mathtt{t}$ *and* $\Pi' \vdash \mathtt{e}$, *then* $\Pi';\cdot \vdash \mathtt{e} : \mathtt{t}$.

The proof for *noninterference* uses a logical relation for modeling the observable equivalence of a well-typed term with respect to an external observer, and shows that the substitutions preserve the equivalence [17]. Space precludes a formal development of the proofs here. Our companion technical report contains the complete rules of our calculus and the full proofs of both the type-safety and the noninterference properties. In addition, the type-safety property of the target language as well as the soundness of the translation in the next subsection are formally specified and mechanically verified[4] in Twelf (a logical framework).

4.2 Translation from $\lambda^{\Pi}_{\le}$ to λ^{Π}_{tag}

Figure 6 shows the translation rules from the typing derivation of a $\lambda^{\Pi}_{\le}$ term $\mathtt{m}$ to a typing derivation of a λ^{Π}_{tag} term $\mathtt{e}$. The main work is in the translation of the subsumption rule which takes the subtyping derivation of the source types and produces a well-typed *coercion function* in the target language. Breazu-Tannen et al. propose [3] such coercion semantics for the subtyping between types in the simply-typed lambda calculus. Our translation slightly extends the semantics for types with labels and permission tags. Our translation is sound as follows:

Theorem 3 (Soundness of permission tagging)

1. *Typing: If* $[\![\Pi;\Gamma \vdash \mathtt{m} : \mathtt{u}]\!] = \mathtt{e}$, *then* $\Pi;[\![\Gamma]\!] \vdash \mathtt{e} : [\![\mathtt{u}]\!]$.
2. *Subtyping: If* $[\![\Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2]\!] = \mathtt{e}$, *then* $\Pi;\Gamma \vdash \mathtt{e} : [\![\mathtt{u}_1]\!] \to [\![\mathtt{u}_2]\!]$.

We conjecture that the translation can be made *coherent* [3], meaning that target terms translated from different typing and subtyping derivations of the *same* source term have the same evaluation behavior. In particular, the

[4]We do not use the higher-order abstract syntax for encoding variable bindings. We have not performed the *totality check* which ensures that all proof cases have been completed — this property is verified externally by hand.

tag checking $\Pi \vdash \mathtt{e}$ performs the same checks whether we tag the function or the argument of an application, hence *coherent* in allowing the same set of legal policy updates. To achieve such coherent translation, *algorithmic subtyping* must be used, instead of *declarative subtyping* as presented in this paper. The conversions and theories between these variants of subtyping are standard [12].

4.3 Discussion

As mentioned in Section 3.1, the fact that a program is noninterfering between updates says nothing of possible information flows across updates. Indeed, in the system described in this section, if an attacker $\mathtt{p}_3$ can observe when updates occur, and what they consist of, it is possible for the timing of an update to communicate a secret value. Consider the following program:

$$\begin{array}{l}
\mathtt{let\ x} = (\mathtt{if\ b}_{\mathtt{p}_1:} \quad (\lambda \mathtt{x{:}bool.\ true}_{\mathtt{p}_1:}) \quad (\lambda[\mathtt{p}_2 \leq \mathtt{p}_1]\mathtt{x{:}bool.}\ [\mathtt{p}_2 :\sqsubseteq \mathtt{p}_1 :]\mathtt{true}_{\mathtt{p}_2:})) \quad \mathtt{in} \\
\mathtt{let\ y} = (\mathtt{if\ (p_2 \leq p_1)} \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in} \\
\mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}
\end{array}$$

Suppose that the program begins evaluating with principal hierarchy $\Pi = \mathtt{p}_2 \leq \mathtt{p}_1$ and that an update $\Pi' = \emptyset$ becomes available just after $\mathtt{x}$ has been computed (call this program p). In the case that $\mathtt{b}$ was $\mathtt{true}_{\mathtt{p}_1:}$ then p would be

$$\begin{array}{l}
\mathtt{let\ x} = \quad \lambda \mathtt{x{:}bool.\ true}_{\mathtt{p}_1:} \quad \mathtt{in} \\
\mathtt{let\ y} = (\mathtt{if\ (p_2 \leq p_1)} \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in} \\
\mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}
\end{array}$$

Thus, the policy update succeeds and $\mathtt{false}_{\mathtt{p}_3:}$ is returned. On the other hand, if $\mathtt{b}$ was $\mathtt{false}_{\mathtt{p}_1:}$, then p is

$$\begin{array}{l}
\mathtt{let\ x} = \quad \lambda[\mathtt{p}_2 \leq \mathtt{p}_1]\mathtt{x{:}bool.}\ [\mathtt{p}_2 :\sqsubseteq \mathtt{p}_1 :]\mathtt{true}_{\mathtt{p}_2:} \quad \mathtt{in} \\
\mathtt{let\ y} = (\mathtt{if\ (p_2 \leq p_1)} \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in} \\
\mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}
\end{array}$$

Thus, the policy update is delayed due to the annotation $\mathtt{p}_2 \leq \mathtt{p}_1$ on the function $\mathtt{x}$ until $\mathtt{z}$ has been evaluated, meaning that $\mathtt{true}_{\mathtt{p}_1:}$ is returned. Hence, $\mathtt{p}_3$ is able to observe $\mathtt{b}_{\mathtt{p}_1}$ even though this is allowed by neither Π or Π'.

This particular example is an artifact of our dynamic tag checking algorithm, since it treats each branch of the initial `if` independently, once evaluated. A more static checking system, suggested earlier, would impose the same constraint on updates whichever function was chosen for `x`, and eliminate this flow. Nonetheless, the noninterference between updates property is too weak to illuminate this issue or its proposed fix, so we plan to consider refinements in future work.

5 Related Work

Security-typed languages for enforcing information flow control are a rich area of research [14]. Security policies are expressed as labels on terms and a principal hierarchy defining delegation relationships; in most systems this hierarchy is fixed at compile-time. Jif [10] and recent formal work [17, 19] support *runtime principals*, which make it possible for the hierarchy to grow at runtime, but do not allow revocations. Our calculus is the first to address generalized, dynamic updates to the principal hierarchy.

Security-type systems are intended to provide a *noninterference* guarantee [13, 4, 5, 9], modulo certain small-bandwidth information channels permitted for performance reasons (timing and termination channels) and an explicit "escape hatch" in the form of a robust downgrading mechanism. The introduction of such downgrading into these languages opened a new chapter in discussions about the meaning of noninterference that is still ongoing [18, 11, 8, 15]. As we have described, our dynamic policy updating is complementary to declassification.

Declassification, as typically used, relabels a data value from one label to another; policy updates as considered in this paper permit the relationship between the labels to change over time. Both features are necessary in practice, and both can potentially be abused—it is possible that work on structured uses of declassification, as provided by robustness [18, 11] or intransitive-noninterference [8] may apply to policy updates as well. We believe our discussion of dynamic policy updating here provides a new avenue for understanding the meaning of noninterference policies for realistic programs.

This work was inspired by a similar system called Proteus that we developed for ensuring type-safety of dynamic software updates [16]. In Proteus, users can define named types $\mathtt{T}$. When given that type $\mathtt{T}= \mathtt{t}$ (for some type $\mathtt{t}$), treating a value of type $\mathtt{T}$ as a $\mathtt{t}$ or vice versa requires an explicit coercion. When a program is dynamically updated to change the definition of $\mathtt{T}$ to be $\mathtt{t}'$, a dynamic analysis can check for these coercions in any functions not being updated (updated functions are assumed compatible with the new definition). If such a coercion is found then the update is only allowed if $\Gamma \vdash \mathtt{t}' \preceq \mathtt{t}$, where Γ is the updated type environment. This dynamic analysis is analogous to dynamic tag checking $\Pi \vdash \mathtt{e}$, which essentially ensures for the new Π that $\Pi \vdash \ell_1 \sqsubseteq \ell_2$ for all tags $[\ell_1 \sqsubseteq \ell_2]$ in $\mathtt{e}$. In Proteus, this dynamic analysis can be replaced with a simpler run-time given certain static information; we conjecture a similar result for λ_{tag}^{Π}.

Primarily in the context of public key infrastructures (PKI), the specific case of credentials revocation has been the subject of considerable study [6, 2]. This work has focused on the exploration of the fundamental tradeoff between security and cost. To simplify, on-line revocation servers effectively permit only a very small window of vulnerability for illicit use of compromised credentials, but often incur a high computation cost. Off-line systems provide a lower computational cost, but do so at the expense of longer latencies for receiving revocation notification. The issue of introducing policy revocation into a running program in a way that maintains sound execution has not been explored in the literature.

6 Conclusion

We have presented a new security-typed language that allows dynamic updating of information-flow policies, in particular the delegation relations in the principal hierarchy. Assumptions needed for sound execution can be represented within the program as *permission tags*, and a run-time tag checking mechanism can be used to prevent illegal updates to the principal hierarchy. Tags are implemented as run-time coercions that capture dynamic labeling behavior, which can prevent spurious rejection of legal policy updates. Tags are added to programs via an automatic translation from a standard source language. We are the first to formalize an information flow language that is sound yet permits dynamic revocations. Our language also satisfies *noninterference between updates*, which seems to us be a reasonable security property in the presence of updates. We hope that our work stimulates interest in making security-typed languages expressive enough to be used in real systems, where policies regularly change.

Acknowledgments We thank Jeff Foster, Peter Sewell, and the anonymous reviewers for their comments. Hicks is supported in part by NSF grant CCF-0346989 (*CAREER: Programming Languages for Reliable and Secure Low-level Systems*). Zdancewic and Tse are supported in part by NSF grant CCR-0311204 (*Dynamic Security Policies*) and NSF grant CNS-0346939 (*CAREER: Language-based Distributed System Security*).

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[2] D. Boneh, X. Ding, G. Tsudik, and M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of USENIX Security Symposium*, pages 297–308, Aug 2001.

[3] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[5] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, April 1984.

[6] Carl A. Gunter and Trevor Jim. Generalized Certificate Revocation. In *ACM Symposium on Principles of Programming Languages*, 2000.

[7] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *ACM Conference on Principles of Programming Languages (POPL)*, 1998.

[8] 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.

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

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

[11] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Asilomar, CA, June 2004. IEEE Computer Society Press.

[12] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] John C. Reynolds. Syntactic control of interference. In *Proc. 5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 39–46, 1978.

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

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

[16] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of theACM Conference on Principles of Programming Languages (POPL)*, January 2005.

[17] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*. IEEE Computer Society Press, May 2004.

[18] 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.

[19] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proceedings of the 2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, Toulouse, France, August 2004.