A Type System for Robust Declassification

Steve Zdancewic[1]

University of Pennsylvania
Department of Computer and Information Science
200 South 33rd Street
Philadelphia, PA 19104-6389

Abstract

Language-based approaches to information security have led to the development of *security type systems* that permit the programmer to describe confidentiality policies on data. Security type systems are usually intended to enforce *noninterference*, a property that requires that high-security information not affect low-security computation. However, in practice, noninterference is often too restrictive—the desired policy does permit some information leakage.

To compensate for the strictness of noninterference, practical approaches include some mechanism for *declassifying* high-security information. But such declassification is potentially dangerous, and its use should be restricted to prevent unintended information leaks. Zdancewic and Myers previously introduced the notion of *robust declassification* in an attempt to capture the desired restrictions on declassification, but that work did not propose a method for determining when a program satisfies the robust declassification condition.

This paper motivates robust declassification and shows that a simple change to a security type system can enforce it. The idea is to extend the lattice of security labels to include *integrity* constraints as well as confidentiality constraints and then require that the decision to perform a declassification have high integrity.

1 Introduction

Security-typed languages track information flow within programs to enforce security properties such as data confidentiality and integrity. Typically, these languages are intended to enforce *noninterference* [17,6], a property that requires that confidential data not affect the publicly visible behavior (outputs, timing, etc.) of a computation.

[1] Email: stevez@cis.upenn.edu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Security properties based on information flow, such as noninterference, provide strong guarantees that confidentiality and integrity are maintained. However, there are several reasons why strict noninterference as a security policy may be undesirable:

- Sometimes the required policy intentionally permits some amount of information flow—confidentiality may conditionally depend on some other factors. For example, Alice may be willing to release her private data to Bob after he pays for it, but not otherwise.
- The policy may require that information be kept secret for only some fixed duration. For example, an on-line auction service must release the value of the winning bid after the auction closes.
- The rate at which information is intentionally leaked may be considered too slow to pose a security threat. For example, a password-checking function of an operating system manages on confidential passwords, but even denying access reveals tiny amounts of information about the correct password.
- Noninterference is the desired policy, but the program analysis is not able to justify the security of some operations. For example, the encryption function of a cryptographic library takes confidential data and makes it public, but the justification that this is secure lies outside the analysis of the security type system.

Consequently, realistic systems include a means of *downgrading*—allowing the security label of the data to be made more public. For confidentiality policies, this process is called *declassification.* However, the ability to escape from the strict confines of noninterference is both essential and dangerous: unregulated use of downgrading can easily result in unexpected release of confidential information.

Because it is potentially dangerous, downgrading should only be used in certain, well-defined ways. One could imagine generalizing information-flow security policies to include specifications of exactly what circumstances permit declassification. The problem with such an approach is that establishing that a given program meets the specifications of the security policy can be extremely difficult—it is the problem of proving that a program meets an arbitrary specification. Moreover, even stating these formal specifications of security policies is hard because they may require an accurate description of a very complex piece of software.

For practicality, we need a natural restriction on the use of declassification in security-typed programming languages. In previous work, Zdancewic and Myers [29] introduced the idea of *robust declassification.* Intuitively, the idea is to limit declassification to be used only when the decision to perform the declassification can be trusted.

As an example, consider a security-typed language implementation of the first scenario in the list above. Alice owns some private data `A`. She is willing to release the data to Bob, but only after he has paid her more than 10 dollars

```
let A:{Alice:} = ... in      // compute Alice's secret
let paid = ... in            // determine amount Bob paid
  if (paid >= 10) then {
    let B:{Alice:Bob} = declassify(A, {Alice:Bob}) in
    // ...Bob can make use of the variable B...
  } else {
    // ...Bob hasn't paid enough, A is secure...
  }
```

Fig. 1. Example use of declassification

for it. The code in Figure 1 shows how this situation might be expressed in a security-typed language.

This program shows how security-typed languages make it possible to program interesting security policies. It first computes Alice's secret, giving the result the *confidentiality label* `{Alice:}`, indicating that she owns the data and that she permits no other readers. This is an example of a security label from Myers' and Liskov's *decentralized label model* [12], which is discussed more in Section 3.

The program next calculates what Bob has paid, and tests whether it is sufficient to satisfy Alice's requirement. If so, the secret is declassified to have label `{Alice:Bob}`, which says that Alice still owns the policy on the data, but that Bob is permitted to read it. Policy decisions like this one are made explicit in the program by requiring that the programmer use the `declassify` operation to mark intentional information leaks.

The issue is that because the declassification reveals Alice's data, she must be careful that this program is invoked appropriately. Furthermore, even when she authorizes this program's use, because the decision to perform the declassification is based in part on the value `paid`, she must trust that `paid` has been computed as described by the program—that is, she must trust the integrity of the data. If Bob were somehow able to maliciously tamper with the contents of the `paid` data or otherwise influence how it's value is computed, he could cause Alice's declassification to be invoked inappropriately.

This problem is exacerbated in a distributed setting, because the computation that determines whether declassification should take place can potentially reside on a different host than the actual declassification itself. Also, if this program appeared as a service running on one host of a distributed system, determining whether another host could invoke the service requires authentication to ensure that Alice's policy is not violated. Much of the motivation for this work comes from experience using the Jif, a security-typed variant of Java [13], in distributed settings [31,32].

The rest of this paper shows how this constraint on the trustworthiness of a decision to declassify data can be formalized in a type system. The idea is to extend the security labels to include *integrity labels* that specify a degree of trust in data. This extension is easily implementable and provides a natural

ℓ, pc	$\in$	$\mathcal{L}$	Security labels
m_ℓ	$\in$	$\mathcal{M}$	Memory cells
t	$::=$	$\texttt{bool}$	Boolean type
	$\mid$	$[\mathsf{pc}]s \to s$	Function type
s	$::=$	t_ℓ	Security types
v	$::=$	$\texttt{t} \mid \texttt{f}$	Boolean base values
	$\mid$	$\lambda x\!:\!s.\, e$	Functions
	$\mid$	x	Variables
e	$::=$	v	Values
	$\mid$	$e\ e$	Function application
	$\mid$	$e \oplus e$	Primitive operations
	$\mid$	m_ℓ	Memory cell contents
	$\mid$	$m_\ell := e$	Imperative update
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	Conditional
$\oplus$	$::=$	$\wedge \mid \vee \mid \ldots$	Boolean operations

Fig. 2. λ_{SEC} grammar

way of thinking about when declassification should be permitted.

The rest of this paper is organized as follows. The next section briefly sketches a typical security-typed language to set the stage for robust declassification. Section 3 describes the decentralized label model and a version with integrity labels. Section 4 discusses how the integrity constraints can be used to implement robust declassification and compares that rule with some alternatives. The paper ends with related work and conclusions.

2 A simple security-typed language

This section briefly sketches a security-typed language called λ_{SEC}, whose grammar is shown in Figure 2. This language is a simplified variant of the SLam calculus, developed by Heintze and Riecke [7]. A full account of this language and several extensions are given in the author's dissertation [28].

In the grammar, ℓ and pc range over elements of a security lattice, $\mathcal{L}$. The elements $\top$ and $\bot$ are the top and bottom elements of $\mathcal{L}$. Points higher in the lattice represent more confidential information, and points lower in the lattice represent more public information. The lattice order and join are written $\sqsubseteq$, and $\sqcup$ respectively.

The possible types include the type `bool` of Boolean values and the types of functions $[\mathtt{pc}]s \rightarrow s$. Security types, ranged over by the metavariable s, are just ordinary types labeled with an element from the security lattice. The security type of a value describes its confidentiality level. Correspondingly, values, v, include the Boolean constants for true and false as well as function values. Expressions include values, primitive Boolean operations such as the logical "and" operation $\wedge$, function application, and a conditional expression.

The language also includes simple imperative features. The state M consists of a globally scoped collection of memory cells that may contain only Boolean values.[2] Each cell m_ℓ has a security label indicating the confidentiality of the data it contains.

For simplicity, we give λ_{SEC} a large-step operational semantics. The evaluation relation is of the form $e, M \Downarrow v, M'$, which means that the (closed) program e evaluates starting in memory state M to a value v and a new memory state M'. The definition of the $\Downarrow$ relation is completely standard, so we omit it.

The type system for λ_{SEC} is designed to prevent unwanted information flows. The basic idea is to associate security-labels with the type information of the program and then take the confidentiality lattice into account when type checking so as to rule out illegal (downward) information flows.

Because upward information flows are allowed (e.g. low-confidentiality data may flow to a high-confidentiality memory cell), the lattice ordering is incorporated as a subtyping relationship [25], written $\vdash s_1 \leq s_2$. The subtyping rules (omitted) establish that $\leq$ is a reflexive, transitive relation that obeys the expected contravariance for function types. In addition, the lattice inequality $\ell \sqsubseteq \ell'$ is lifted to subtyping: $\vdash t_\ell \leq t_{\ell'}$. This eliminates the need for the programmer to make explicit when information flows are permissible. For example, we can conclude $\vdash \mathtt{bool}_\perp \leq \mathtt{bool}_\top$ because anywhere a high-security Boolean can be safely used, a low-security Boolean can also be used. Intuitively, if the program is sufficiently secure to protect high-security data, it also provides sufficient security to "protect" low-security data.

There is a subtlety with security types in the presence of mutable state: There can be implicit flows that leak information about control flow into the state. Consider the following program, where $m_\perp$ is a memory cell that contains low-security values and h is a variable of type $\mathtt{bool}_\top$.

$\mathtt{if}\ h\ \mathtt{then}\ m_\perp\ \mathtt{:=}\ \mathtt{t}\ \mathtt{else}\ m_\perp\ \mathtt{:=}\ \mathtt{f}$

Here, the problem is that, even though the assignments to $m_\perp$ are individually secure, which of them occurs depends on high-security data. To prevent such implicit flows, the type system for λ_{SEC} associates a label `pc` with the program counter. Intuitively, the program counter label approximates the information that can be learned by observing that the program has reached a particular

[2] Other work [30,16] shows how to treat more complex stores that may contain structured data and functions.

True $$\Gamma\ [\textsf{pc}] \vdash \texttt{t} : \texttt{bool}_{\textsf{pc}}$$

False $$\Gamma\ [\textsf{pc}] \vdash \texttt{f} : \texttt{bool}_{\textsf{pc}}$$

Var $$\frac{\Gamma(x) = s}{\Gamma\ [\textsf{pc}] \vdash x : s \sqcup \textsf{pc}}$$

Fun $$\frac{\Gamma, x\!:\!s_1\ [\textsf{pc}_1] \vdash e : s_2 \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma\ [\textsf{pc}_2] \vdash \lambda x\!:\!s_1.\, e : ([\textsf{pc}_1]s_1 \to s_2)_{\textsf{pc}_2}}$$

Binop $$\frac{\Gamma\ [\textsf{pc}] \vdash e_1 : \texttt{bool}_{\ell_1} \quad \Gamma\ [\textsf{pc}] \vdash e_2 : \texttt{bool}_{\ell_2}}{\Gamma\ [\textsf{pc}] \vdash e_1 \ \oplus\ e_2 : \texttt{bool}_{(\ell_1 \sqcup \ell_2)}}$$

App $$\frac{\Gamma\ [\textsf{pc}_1] \vdash e_1 : ([\textsf{pc}_2]s_2 \to s)_\ell \quad \Gamma\ [\textsf{pc}_1] \vdash e_2 : s_2 \quad \ell \sqsubseteq \textsf{pc}_2}{\Gamma\ [\textsf{pc}_1] \vdash e_1\ e_2 : s \sqcup \ell}$$

Cond $$\frac{\Gamma\ [\textsf{pc}] \vdash e : \texttt{bool}_\ell \quad \Gamma\ [\textsf{pc} \sqcup \ell] \vdash e_i : s \quad i \in \{1,2\}}{\Gamma\ [\textsf{pc}] \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : s}$$

Deref $$\frac{}{\Gamma\ [\textsf{pc}] \vdash m_\ell : \texttt{bool}_\ell \sqcup \textsf{pc}}$$

Assign $$\frac{\Gamma\ [\textsf{pc}] \vdash e : \texttt{bool}_{\ell'} \quad \ell' \sqsubseteq \ell}{\Gamma\ [\textsf{pc}] \vdash m_\ell := e : \texttt{bool}_{\textsf{pc}}}$$

Sub $$\frac{\Gamma\ [\textsf{pc}] \vdash e : s \quad \vdash s \leq s' \quad \textsf{pc}' \sqsubseteq \textsf{pc}}{\Gamma\ [\textsf{pc}'] \vdash e : s'}$$

Fig. 3. Typing λ_{SEC}

point during the execution. In the example above, the program counter reveals the value of h, so inside the branches of the conditional, we have $\mathsf{pc} = \top$. To prevent these implicit flows, the labeled semantics requires that $\mathsf{pc} \sqsubseteq \ell$ whenever an assignment to a reference m_ℓ occurs in the context with program counter label pc. This rules out the above example.

Another implicit information flow can arise due to the interaction between functions and state. For example, consider a function f that takes a Boolean but only assigns the location $m_\bot$ the constant $\mathtt{t}$. Function f is defined as:

$$f \overset{\text{def}}{=} \lambda x\!:\!\mathtt{bool}_\top.\, m_\bot := \mathtt{t}$$

This function is perfectly secure and can be used in many contexts, but it can also be used to leak information. For example, consider the program below:

```
m_ℓ := f;
if h then f t else t;
```

This program is insecure because f writes to the low-security memory location m_ℓ, but whether f is invoked depends on secret data h. In general, calls to functions that have side effects (in this case, writes to memory locations) can leak information in the same way that assignment leaks information about the program counter.

To detect and rule out such implicit flows, the type system includes *effects* in the style of Jouvelot and Gifford [9]. Function types in λ_{SEC} include an additional label; they are of the form $[\mathsf{pc}]s_1 \to s_2$. The label pc is a lower bound on the labels of any locations that might be written when calling the function. To call a function of this type in a context where the program counter has label pc', the type system requires that $\mathsf{pc}' \sqsubseteq \mathsf{pc}$. In the example above, because f writes to a low security location, f is given the type $[\bot]\mathtt{bool}_\top \to \mathtt{bool}_\bot$; however, since $\mathsf{pc} = \top$ inside the branches of the conditional guarded by h, the call to f in the above program is ruled out.

These intuitions guide the design of λ_{SEC}'s type system, the rules for which are given in Figure 3. They are judgments of the form $\Gamma\ [\mathsf{pc}] \vdash e : s$, which says "under the assumptions provided by Γ, the term e is a secure program that evaluates to a value of type s and does not modify memory cells with label strictly less than pc."

Although space prohibits full explanation of the type system (see elsewhere [28] for the details), note that λ_{SEC} enjoys the usual type soundness theorems and, more importantly, we can establish the following noninterference result.

Theorem 2.1 (Noninterference) *Suppose $\ell \not\sqsubseteq \zeta$. If $x : t_\ell\ [\bot] \vdash e : \mathtt{bool}_\zeta$ and $\vdash v_1, v_2 : t_\ell$, then for all M*

$$e\{v_1/x\}, M \Downarrow v, M_1 \iff e\{v_2/x\}, M \Downarrow v, M_2$$

where $M_1 \approx_\zeta M_2$.

The notation $M_1 \approx_\zeta M_2$ means that memories M_1 and M_2 agree on cells whose labels are $\sqsubseteq \zeta$:

$$M_1 \approx_\zeta M_2 \overset{\text{def}}{=} \forall m_\ell.\ (\ell \sqsubseteq \zeta) \Rightarrow M_1(m_\ell) = M_2(m_\ell)$$

Thus, the noninterference theorem says that the low-security results (with label $\sqsubseteq \zeta$) computed by a program deemed secure by the type system are not affected by altering the value of high-security data (with label $\not\sqsubseteq \zeta$).

Note that λ_{SEC}, as described so far, does not include a declassification mechanism. Clearly the noninterference theorem will not hold if declassification is added—the difficulty is in trying to establish properties related to noninterference that *do* hold in the presence of declassification.

3 The decentralized label model

One strategy for dealing with declassification is to limit its use to particular parts of the program. The *decentralized label model* (DLM) proposed by Myers and Liskov [12] adds additional structure to the security lattice in order to do exactly that.

Central to the DLM is the notion of a *principal*, which is an entity (e.g., user, process, party) that can have a confidentiality or integrity concern with respect to data. Principals own information-flow policies and are also used to define the *authority* possessed by the running program. The authority A at a point in the program is a set of principals that are assumed to authorize any action taken by the program at that point—in particular, principals may authorize declassifications of data. Different program points may have different authority, which must be explicitly granted by the principals in question.

A simple confidentiality label in this model is written $\texttt{\{o:r}_1\texttt{,r}_2\texttt{,}\ldots\texttt{,r}_n\texttt{\}}$, meaning that the labeled data is owned by principal `o`, and that `o` permits the data to be read by principals $\texttt{r}_1$ through $\texttt{r}_n$ (and, implicitly, `o`). Such a policy is sometimes abbreviated $\texttt{\{o:}\vec{\texttt{r}}\texttt{\}}$, where $\vec{\texttt{r}}$ is the set of readers for the policy.

Data may have multiple owners, each controlling a different component of its label. For example, the label $\texttt{\{o}_1\texttt{:r}_1\texttt{,r}_2\texttt{; o}_2\texttt{:r}_1\texttt{,r}_3\texttt{\}}$, contains two components and says that owner $\texttt{o}_1$ allows readers $\texttt{r}_1$ and $\texttt{r}_2$ and owner $\texttt{o}_2$ allows readers $\texttt{r}_1$ and $\texttt{r}_3$. The interpretation is that *all* of the policies described by a label must be obeyed, only $\texttt{r}_1$ will be able to read data with this annotation. Such composite labels arise naturally in collaborative computations: for example, if `x` has label $\texttt{\{o}_1\texttt{:r}_1\texttt{,r}_2\texttt{\}}$ and `y` has label $\texttt{\{o}_2\texttt{:r}_1\texttt{,r}_3\texttt{\}}$, then the sum `x + y` has the composite label $\texttt{int\{o}_1\texttt{:r}_1\texttt{,r}_2\texttt{; o}_2\texttt{:r}_1\texttt{,r}_3\texttt{\}}$, which expresses the conservative requirement that the sum is subject to both the policy on `x` and the policy on `y`.

In the lattice, $\ell_1 \sqsubseteq \ell_2$ if the label ℓ_1 is less restrictive than the label ℓ_2. Data with label ℓ_1 is less confidential than data with label ℓ_2—more principals are permitted to see the data, and, consequently, there are fewer restrictions on how data with label ℓ_1 may be used. For example, $\texttt{\{o:r\}} \sqsubseteq \texttt{\{o:\}}$ holds because the left label allows both `o` and `r` to read the data, whereas the right label admits only o as a reader. The bottom of the DLM confidentiality lattice is the label $\bot = \{\}$ and, when there are n principals $\texttt{o}_1$ through $\texttt{o}_n$, the top of the lattice is the label $\top = \{\texttt{o}_1\texttt{:;}\ldots\texttt{;o}_n\texttt{:}\}$—all principals claim sole ownership of the data, so none may read it.

The full definition of $\sqsubseteq$ for the decentralized label model is given in Myers' thesis [11]. His thesis also shows that $\sqsubseteq$ is a pre-order whose equivalence classes form a distributive lattice. The label join operation combines the restrictions on how data may be used. As an example, $\{\texttt{o:r}_1\texttt{,r}_2\} \sqcup \{\texttt{o:r}_1\texttt{,r}_3\} = \{\texttt{o:r}_1\}$, which includes the restrictions of both labels

3.1 Integrity constraints

Integrity constraints are the dual to confidentiality constraints. A confidentiality policy specifies where information may flow *to*, whereas an integrity policy specifies where information may flow *from.*

Integrity policies can also be expressed using the security lattice approach: high-integrity data has fewer restrictions on how it should be used so should have a label lower in the security lattice than low-integrity data. Thus, the simplest non-trivial integrity lattice consists of two labels $\top$, the high-integrity label, and $\bot$, the low-integrity label, but their order is the opposite from the usual confidentiality label: $\top \sqsubseteq \bot$.

One natural interpretation of an integrity label is a set of principals that trust the value of a piece of data with the corresponding label. We can extend the standard confidentiality model with components that specify these simple integrity constraints. The label $\{\texttt{*:p}_1\texttt{,}\ldots\texttt{,p}_n\}$ specifies that principals $\texttt{p}_1$ through $\texttt{p}_n$ *trust* the data—they believe the data to be computed by the program as written.

In this approach, integrity policies have no owner—the notation "*" is used to suggest that the owner of the policy does not matter. With this definition, the integrity label {*:} specifies a piece of data trusted by no principals; it is the label of completely untrusted data and hence the top of the integrity lattice. Conversely, if there are n principals $\texttt{p}_1$ through $\texttt{p}_n$, the bottom of the lattice is the label $\bot = \{\texttt{*:p}_1\texttt{,}\ldots\texttt{,p}_n\}$. Data with this label is universally trusted.

This is a weak notion of integrity; it specifies only which principals trust the data, not how the data may be modified or what invariants that high-integrity data must satisfy. However, it is sufficient for the purposes of explaining robust declassification. One can easily generalize this idea to a richer form that is more fully dual to the DLM owners–readers model by specifying an owners and writers of the data [12].

Note that the combined confidentiality and integrity lattice can be constructed by taking the product of the two lattices. Such labels combine integrity and confidentiality components, and they also arise naturally when computing with many sources of information. For the these extended labels, the functions $C(\ell)$ and $I(\ell)$ respectively extract the confidentiality and integrity parts of ℓ.

4 Declassification

We can now see how the decentralized label model attempts to control the use of declassification. First consider the simplest way to add a declassification operation to a security-typed language. We extend the syntax:

$$e ::= \dots \quad | \quad \texttt{declassify(}e, \ell\texttt{)}$$

The $\texttt{declassify(}e, \ell\texttt{)}$ expression downgrades the security label in the type of e to ℓ. To reflect this interpretation, we add the following typing judgment.

$$\textsc{Bad-Declassify} \qquad \frac{\Gamma\ [\mathsf{pc}] \vdash e : t_{\ell'}}{\Gamma\ [\mathsf{pc}] \vdash \texttt{declassify(}e, \ell\texttt{)} : t_{\ell}}$$

This judgment says that a value v with an arbitrary label can be given any other arbitrary label by declassification. It clearly breaks noninterference because high-security data can now be made low-security. Declassification is intended for this purpose, but this rule is too permissive—it can be used at any point to release confidential information. Consequently, adding such a rule to λ_{SEC} completely invalidates its noninterference theorem. We get no guarantees about the security of programs that use declassification, and the program may as well have been written without security types.

The decentralized label model introduces the notion of *authority* to allow coarse-grained control over where declassifications may be used. The idea is to associate a set of principals, the code's authority, with each portion of the program. Because the DLM labels include information about which principals own the policies on the data, it is straightforward to determine which principals' policies are being weakened by a given declassify operation. A principal `p`'s authority is needed to perform declassifications of data owned by `p`. For example, owner `o` can add a reader `r` to a piece of data `x` by declassifying its label from `{o:}` to `{o:r}` using the expression `declassify(x, {o:r})`.

We use the function $\mathsf{auth}(\ell, \ell')$ to determine the set of principals whose authority is needed to move from label ℓ to label ℓ' in the lattice. For example, $\mathsf{auth}(\texttt{\{o:\}}, \texttt{\{o:r\}}) = \{\texttt{o}\}$. In general, when $C(\ell) = \texttt{\{}\mathtt{o}_1\texttt{:}\vec{\mathtt{r}}_1\texttt{;}\dots\texttt{;}\mathtt{o}_n\texttt{:}\vec{\mathtt{r}}_n\texttt{\}}$ and $C(\ell') = \texttt{\{}\mathtt{o}'_1\texttt{:}\vec{\mathtt{r}}'_1\texttt{;}\dots\texttt{;}\mathtt{o}'_m\texttt{:}\vec{\mathtt{r}}'_m\texttt{\}}$, the definition of required authority is:

$$\mathsf{auth}(\ell, \ell') = \{\ \mathtt{o} \mid \mathtt{o} = \mathtt{o}_i = \mathtt{o}'_j \quad \wedge \quad \vec{\mathtt{r}}'_j \supset \vec{\mathtt{r}}_i\}$$

To incorporate the DLM notion of authority into the type system, we extend the security part of the typing context to include A, the set of principals that have authorized the program being checked. This gives DLM typing judgments the form $\Gamma\ [\mathsf{pc}, A] \vdash e : s$. We can now give the DLM rule for declassification.

$$\textsc{DLM-Declassify} \qquad \frac{\Gamma\ [\mathsf{pc}, A] \vdash e : t_{\ell} \quad \mathsf{auth}(\ell, \ell') \subseteq A}{\Gamma\ [\mathsf{pc}, A] \vdash \texttt{declassify(}e, \ell'\texttt{)} : t_{\ell'}}$$

In addition, the type of this DLM function should reflect the authority needed to invoke the function, just as λ_{SEC}'s function types include the pc from the security context. Therefore, the typing rule for functions is:

$$\text{DLM-Fun}\qquad \frac{\Gamma, x\!:\!s'\ [\mathsf{pc}', A'] \vdash e : s}{\Gamma\ [\mathsf{pc}, A] \vdash \lambda x\!:\!s'.\ e : ([A', \mathsf{pc}']s' \to s)_{\mathsf{pc}}}$$

The function type $[\mathsf{pc}', A]s' \to s$ indicates that the function requires authority A and so may perform declassifications on behalf of the principals in A. The programmer can delimit where declassifications may take place by constraining the types assigned to the possible clients of a given function.

The caller of a function must establish that it has the authority necessary to carry out any of the declassifications that might occur inside the function call. This requirement is reflected in the function application rule:

$$\text{DLM-App}\qquad \frac{\begin{array}{l}\Gamma\ [\mathsf{pc}, A] \vdash e : ([\mathsf{pc}', A']s' \to s)_\ell \\ \Gamma\ [\mathsf{pc}, A] \vdash e' : s' \\ A' \subseteq A \quad \ell \sqsubseteq \mathsf{pc}'\end{array}}{\Gamma\ [\mathsf{pc}, A] \vdash e\ e' : s}$$

Under this model, the example program from Figure 1 would require `Alice`'s authority to run, because `auth`(`{Alice:}`, `{Alice:Bob}`) is the set {`Alice`}.

One could imagine using authority in other ways. For example, it would be easy to adjust this type system to allow functions to be endowed with some fixed authority. This approach would allow some of the authority to come from a function's calling context and some authority to belong to the function itself.

The benefit of this approach is that whenever a program is well typed under a security context with authority A, it is guaranteed not to leak confidential information owned by principals not in A.

4.1 Robust declassification

Despite the increased control of downgrading offered by the decentralized label model, there is a weakness in its simple, authority-based approach. The problem is illustrated once again by the example in Figure 1. Even though Alice's authority is necessary for the declassification to be carried out, it is not sufficient to ensure that her security policy, as encoded in the program is not violated. The problem is that the decision to perform the declassification is affected by the contents of `paid`, which may not be trusted by Alice.

Rather than give authority to the entire function body, it seems more natural to associate the required authority with the *decision* to perform the declassification. The program-counter label at the point of a `declassify` expression is already a model of the information used to reach the declassification. Therefore, to ensure that the decision to do the declassification is sufficiently trusted, we simply require that the program counter have high enough integrity.

These intuitions are captured in the following rule for declassification:

$$\textsc{Robust-Declassify}\quad \frac{\Gamma\ [\mathsf{pc}] \vdash e : t_{\ell} \quad I(\ell) = I(\ell') \quad I(\mathsf{pc}) \sqsubseteq \texttt{\{*:}\mathsf{auth}(\ell,\ell')\texttt{\}}}{\Gamma\ [\mathsf{pc}] \vdash \texttt{declassify(}e,\ell'\texttt{)} : t_{\ell'}}$$

This approach equates the authority of a piece of code with the integrity of the program counter at the start of the code, simultaneously simplifying the typing rules—no authority context A is needed—and strengthening the restrictions on where declassification is permitted.

This version of declassification rules out the program in Figure 1. To allow this program to typecheck, the programmer would be forced to add Alice's integrity constraint `{*:Alice}` to the variable `paid`. Doing so would force the calculation of paid to depend only on data that Alice deems trustworthy.

The benefit of tying declassification to integrity is that the noninterference proofs given for the security-typed language say something meaningful for programs that include declassification. Note that the declassification operation does not change the integrity of the data being declassified. Projecting the noninterference result onto the integrity sublattice yields the following lemma as a corollary. It is a weak guarantee: Intuitively, low-integrity data cannot interfere with what data is declassified.

Lemma 4.1 (Robust Declassification) *Suppose that $x : s[\bot] \vdash e : s'$ and the integrity labels satisfy $I(\mathsf{label}(s)) \not\sqsubseteq I(\mathsf{label}(s'))$. Then for any values v_1 and v_2 such that $\vdash v_i : s$ it is the case that $e\{v_1/x\} \Downarrow v \iff e\{v_2/x\} \Downarrow v$.*

This lemma holds regardless of whether e contains declassification operations. This lemma does not say anything about what high-security information might be declassified. Nevertheless, it is better than giving up all security properties when declassifications are used. Moreover, using the integrity constraint still implies the property guaranteed by the DLM authority model: If $\Gamma\ [\mathsf{pc}] \vdash e : s$ and $I(\mathsf{pc}) \not\sqsubseteq$ `{*:p}` then e cannot contain any declassifications on `p`'s behalf.

One could generalize robust declassification by associating with each distinct declassification expression in the program a separate principal d and requiring that $I(\mathsf{pc}) \sqsubseteq$ `{?:d}` in the declassification typing judgment. This constraint allows the programmer to name particular declassifications in security policies so that, for instance a value with integrity label $\texttt{\{?:d}_1\texttt{, d}_2\texttt{\}}$ could possibly be declassified at points $\mathtt{d}_1$ and $\mathtt{d}_2$ but not at a declassification associated with point $\mathtt{d}_3$ in the program.

Whether such a generalization would be useful in practice, and how to precisely characterize the confidentiality properties of the resulting programs remains for future work.

5 Related work

There has been much recent work in security-typed languages, ranging from simple calculi [25,7,1,21,24,30,23,8] to full-featured languages [10,31,16,2]. For a recent survey of this work, see Sabelfeld and Myers' paper [20].

The simplest and most standard approach to declassification is to restrict its uses to those performed by a trusted subject, similar to the DLM requirement that a function possess the proper authority. This approach does not address the question of whether an information channel is created. Many systems have incorporated a more limited form of declassification. Ferrari et al. [4] augment information flow controls in an object-oriented system with a form of dynamically-checked declassification called *waivers*. However, these efforts provide only limited characterization of the safety of declassification.

Pottier and Conchon [15] argue that access control and information flow are orthogonal issues, and that the use of declassification mechanisms can be moderated by access controls. This proposal seems similar to Jif's use of authority declarations. Whether access control alone is sufficient in practice to regulate declassification remains for future research.

The interplay between authority and declassification is similar to Java's stack inspection security model [27,26,5]. In Java, privileged operations can require that they be invoked only in the context of some authorization clause, and, that, dynamically, no untrusted methods are between the authorization and the use of the privileged operation on the call stack. These constraints on the run-time stack are similar to the authority constraints used in the decentralized label model, but weaker than the robust declassification mechanism proposed here. The difference is that the stack-inspection approach does not track the integrity of data returned by an untrusted piece of code, so untrusted data might still influence privileged operations.

Using untrusted data to regulate privileged operations is related to buffer overflow bugs found in C programs. The C libraries assume that strings are properly delimited and do not check their bounds. (The libraries assume that the strings have high integrity.) Programs use the libraries without appropriate checks, making them vulnerable to using strings read from an untrusted source such as the network. Analyses that find such format string vulnerabilities in C [22] are similar to integrity-only information-flow analysis.

Intransitive noninterference policies [19,14,18] generalize noninterference to describe systems that contain restricted downgrading mechanisms. The work by Bevier et al. on *controlled interference* [3] is most similar to this work in allowing policies for information released to a set of *agents*.

6 Conclusions

Security-typed languages are a promising and flexible approach to the problem of protecting confidential data in computer systems. Practical security-typed languages provide a rich vocabulary for building security policies, which are enforced by static program analyses. Although strict noninterference is a valuable starting point for developing information flow policies, it is sometimes necessary to violate noninterference in practice. Therefore, some form of downgrading mechanism is required.

This paper presents a type system for robust declassification, which is an attempt to control improper use of downgrading. The main intuition is that the decision to perform a declassification must be trusted by any principals whose security policies are weakened by the declassification. This insight leads naturally to an integrity constraint on the data used to make the decision. In addition, simple modifications to existing security type systems can implement robust declassification.

Although this approach is promising, there is still further research necessary to understand how downgrading mechanisms affect information-flow security policies. For instance, this paper has focused exclusively on declassification, but once integrity constraints are included it is natural to ask how declassification's integrity counterpart (called *endorsement*) should be similarly constrained. In addition, it is not clear how downgrading mechanisms should interact with polymorphism and mechanisms for describing dynamic security policies.

6.1 Acknowledgments

Andrew Myers and Andrei Sabelfeld provided insightful observations and engaging discussion in our quest to understand declassification. Many thanks to Stephanie Weirich for her feedback on earlier drafts of this paper.

References

[1] 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.

[2] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[3] William R. Bevier, Richard M. Cohen, and William D. Young. Connection policies and controlled interference. In *Proc. of the 8th IEEE Computer Security Foundations Workshop*, pages 167–176, 1995.

[4] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 1997.

[5] Cedric Fournet and Andrew Gordon. Stack inspection: Theory and variants. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–318, 2002.

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

[7] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[8] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 81–92, January 2002.

[9] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

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

[11] 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.

[12] 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.

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

[14] Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symposium on Security and Privacy*, 1995.

[15] 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.

[16] 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.

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

[18] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[19] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

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

[21] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[22] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[23] Geoffrey Smith. A new type system for secure information flow. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, pages 115–125. IEEE Computer Society Press, June 2001.

[24] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, January 2000.

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

[26] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[27] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1998.

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

[29] 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.

[30] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2/3), 2002.

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

[32] 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, 2003.