Downgrading Policies and Relaxed Noninterference

Peng Li
University of Pennsylvania
lipeng@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

ABSTRACT

In traditional information-flow type systems, the security policy is often formalized as noninterference properties. However, noninterference alone is too strong to express security properties useful in practice. If we allow downgrading in such systems, it is challenging to formalize the security policy as an extensional property of the system.

This paper presents a generalized framework of downgrading policies. Such policies can be specified in a simple and tractable language and can be statically enforced by mechanisms such as type systems. The security guarantee is then formalized as a concise extensional property using program equivalences. This *relaxed noninterference* generalizes traditional pure noninterference and precisely characterizes the information released due to downgrading.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints, Data types and structures, Frameworks*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification Techniques, Invariants, Mechanical verification*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection.

General Terms

Languages, Design, Security, Theory.

Keywords

Downgrading policies, information flow, language-based security, relaxed noninterference, program equivalence.

1. INTRODUCTION

The Challenge of Downgrading

In this paper we focus on a specific area of computer security research, namely, language-based information-flow security [17], where the target systems are computer programs. The security properties we care about are confidentiality and integrity, specified by information-flow policies, which are usually formalized as *noninterference* [4] [8], a global extensional property of the program that requires that confidential data not affect the publicly visible behavior. Such information flow policies can be enforced by mechanisms like type systems and static program analysis [19] [13] [14] [1].

In information-flow control, each piece of data is annotated by a *label* that describes the security level of the data. Such labels usually form a partially ordered set. Pure noninterference policies only allow data flow from low security places to high security places. As a program runs, the label of data can only become higher. This restriction is not practical in most applications. Take the example of a login process, the password is a secret and it has a higher security level than the user-level data. By comparing the user input with the password and sending the result back to the user, data flows from high to low, thus the noninterference property is violated.

We use the word *downgrading* to specify information flow from a high security level to low security level. It is also called *declassification* for confidentiality and *endorsement* for integrity. As we allow downgrading in the system, pure noninterference no longer holds and the security policy of the whole system becomes much more complex. Instead of using an elegant extensional property such as noninterference, most downgrading policies are intensional, specifying exactly what circumstances information can flow in which order. To formally specify such policies, we may require an accurate description of these intensional properties in a complex piece of software, which can be very complicated. Such security policies can be hard to specify, understand and enforce. It is also difficult to prove the soundness of the corresponding enforcement mechanism.

Our Contribution

We approach the downgrading problem by allowing the user to specify *downgrading policies*. We use a type system to enforce such policies, and formalize the security goal as an extensional property called *relaxed noninterference*, which generalizes pure noninterference and accurately describes the effects due to downgrading. Our research is based on the observation that a noninterfering program $f(h, l)$ can usually be factored to a "high security" part $f_{\mathsf{H}}(h, l)$ and a "low security part" $f_{\mathsf{L}}(l)$ that does not use any of the high-level inputs h. As a result, noninterference can be proved by transforming the program into a special form that does not depend on the high-level input. Relaxed noninterference can then be formalized by factoring the program into the

POPL'05, January 12–14, 2005, Long Beach, California, USA.

composition of such a special form and some functions that depend on the high-level inputs, which we treat as downgrading policies.

2. BACKGROUND AND RELATED WORK

Before presenting our results in detail, it is useful to describe some prior approaches to the problem of downgrading.

DLM and Robust Declassification

The *decentralized label model* (DLM) invented by Myers and Liskov [10] puts access control information in the security labels to specify the downgrading policy for the annotated data. Different mutually-distrusting principals can specify their own access control rules in the same label. Such labels are well-structured and can be used to express both confidentiality and integrity. Downgrading is controlled based on the code authority and the access control information in the label of data to be downgraded: each principle can only weaken its own access control rules. Practical languages such Jif [9] have been built based on the DLM.

The downgrading policy specified by the DLM is highly intensional and it is difficult to formalize as an extensional property of the program. Once downgrading happens in the program, the *noninterference* property is broken and the user cannot reason about the effects of downgrading. Trusted code can downgrade its data in arbitrary ways, whereas untrusted code cannot downgrade any data that does not belong to it.

Robust declassification [20] improves the DLM by imposing a stronger policy on downgrading that requires the decision to perform downgrading operations only depend on trustworthy (high-integrity, untainted) data. Such a policy can be formalized and the security guarantee can be expressed as an extensional property of the system [11]. Nevertheless, it only addresses one particular useful policy for downgrading. It cannot provide detailed guarantees on how the data is downgraded, and downgrading is still forbidden for untrusted code.

Our work borrows some philosophy from robust declassification. Although we are concerned with confidentiality and the process of declassification, the policies for downgrading can be thought of as integrity properties of the system: they require the downgrading operation to be trustworthy and correct with respect to some specification.

Complexity Analysis and Relative Secrecy

To look for a system-wide extensional guarantee with the existence of downgrading, Volpano and Smith proposed the *relative secrecy* [18] approach as a remedy for noninterference. They designed a type system that contains a *match* primitive, where the secret can only be leaked by comparing it to untrusted data via this primitive. The security goal is then formalized as a computational complexity bound of the attack.

However, this approach lacks some flexibility in practical applications. It assumes that there is a single secret in the system and the attack model for the system is fixed, thus it only enforces one particular useful downgrading policy using a particular mechanism. To express and enforce other downgrading policies like "the parity of the secret integer n can be leaked", we need completely different frameworks and mechanisms.

Abstract Noninterference

Giacobazzi and Mastroeni used abstract interpretations to generalize the notion of noninterference by making it parametric to what the attacker can analyze about the information flow [3]. Many downgrading scenarios can be formally characterized in this framework, and the security guarantee is formalized in a weakened form of noninterference. However, this framework is mainly theoretical. To practically apply this theory in building program analysis tools, we need to design ways to express the security policies and mechanisms to enforce such policies.

Intransitive Noninterference

Our work has a close relationship to *intransitive noninterference* [15] [7], where special downgrading paths exist in the security lattice. During downgrading, data can flow indirectly through these paths, although there is no direct lattice ordering between the source and the destination. We improve this idea of *intransitive noninterference* by parameterizing the downgrading paths with actions, and globally reasoning about the effects due to downgrading.

Quantifying Information Flow

Some interesting work has been done using quantitative approaches [5] [6] [12] to precisely estimate the amount of information leakage when downgrading is available. Drawing on this research, we order the security levels by comparing their *abilities* to leak information. Programs leaking more information are considered less secure. However, comparing the quantity of information leakage does not have directly sensible meanings in many situations. Instead of using real numbers as metrics for information leakage, we use program fragments; the information order is defined among these programs.

Delimited Information Release

Sabelfeld and Myers [16] proposed an end-to-end security guarantee called *delimited release* that generalizes noninterference by explicitly characterizing the computation required for information release. Our work generalizes *delimited release* in two ways. First, we treat the computation required for declassification as security policies and use these policies to represent security levels for each piece of data in the system. Second, downgrading can be fine-grained and implicit in our framework. We formalize the security guarantee by transforming a safe program to the form of *delimited release*, where all the downgrading expressions explicitly match the downgrading policies.

3. A FRAMEWORK OF DOWNGRADING POLICIES

3.1 The Motivation

The focus of our research is studying *downgrading policies*. Instead of studying *"who can downgrade the data"* as the decentralized label model did, we take an orthogonal direction and study *"how the data can be downgraded"*. Instead of having various mechanisms that provides vastly different kinds of security guarantees, we would like to have a more general framework where the user can specify *downgrading policies* that accurately describes their security requirement, and have the enforcement mechanism carry out such policies. We have the following goals for downgrading policies:

Expressiveness: The programmers should be able to specify a rich set of downgrading policies depending on their highly-customized security requirement. Such policies are fine-grained and describes security requirements for each piece of data in the system. For example: some data is a top secret and we do not allow any information to leak from it; some secrets can be downgraded by encrypting them; for some secret data we can safely reveal the lowest several bits; root passwords can only be leaked by comparing them to public data; etc.

Representability: The downgrading policies should be formally specified in representable forms. It should be easy for the programmer to write down their policies and such policies are meant to be understood by both human and machines. In this paper, we use a simple programming language to express downgrading policies and treat these policies as security levels so that the programmer can use them as type annotations.

Tractability: Such policies must be enforceable by some mechanisms such as type systems or model checking. Since we are extending the traditional language-based information-flow security, it is desirable to use similar *static* approaches, where the policies are enforced at compilation time rather than at run time.

Extensional Guarantee: This is the main challenge we face: if the policies are enforced by some mechanism, what are the security guarantees they bring to the user? The policies are fine-grained and the enforcement mechanisms are usually *intensional*, yet we would like to have a formal, system-wide, *extensional* security guarantee that looks simple, elegant, understandable and trustworthy. We also want to formally prove the soundness of the enforcement mechanism with respect to this security guarantee. In this paper, we express such guarantees in a form of *relaxed noninterference*, where the effects of downgrading policies can be accurately characterized by program equivalences.

3.2 Downgrading Policies as Security Levels

The main idea of our framework is to treat downgrading policies as security levels in traditional information flow systems. Instead of having only H and L in the security lattice, we have a much richer lattice of security levels where each point in the lattice corresponds to some downgrading policy, describing how the data can be downgraded from this level. For example, the policy corresponding to H is that the information cannot be leaked to public places by any means, whereas the policy implied by L is that the data can be freely leaked to the public places. In our policy language, we express H using constant functions and express L using identity functions.

The security levels in the middle of the lattice are more interesting. We take the following program as an example, where the security policy for `secret` is that "the secret can only be leaked by comparing the lowest 64 bits of its hashed value to some public data", and `input,output` have security level L.

Example 3.2.1 (Downgrading).

```
x := hash(secret);
y := x % 2^64;
if (y=input) then output:=1 else output:=0;
z := x % 3;
```

Downgrading happens when the secrets are involved in some computation. In the first statement, we computed the hash of the secret, so the downgrading policy for `x` should be that "`x` can only be leaked by comparing its lowest 64 bits to some public data". After the second statement, the policy for `y` should be that "`y` can only be leaked by comparing it to some public data". In the branching statement, the policy for the conditional `(y=input)` should be L because `y` is compared to `input`. Therefore, the information leak from `secret` to `output` is safe with respect to the downgrading policy of `secret`. However, in the last statement, we cannot find a way to downgrade `z` while satisfying the policy for `x` and `secret`. To be safe, the security level for `z` can only be H: it cannot be downgraded by any means.

With the existence of downgrading, the ordering among these security levels is more complicated than in the traditional security lattice. Briefly speaking, there are two kinds of ordering here.

- Subtyping order. We can extend the traditional $\mathsf{L} \sqsubseteq \mathsf{H}$ lattice with something in the middle: $\mathsf{L} \sqsubseteq l_s \sqsubseteq \mathsf{H}$ where l_s denotes the security level of `secret`. We can see that it is always safe for information to flow from lower levels to higher levels, because it is equivalent to making the downgrading policy more restrictive. However, the security level l_x for `x` has no such ordering with l_s because it does not make sense to give `x` the same downgrading policy as `secret` — doing so will violate the downgrading policy for `secret`.

- Downgrading order. Although we do not have $l_x \sqsubseteq l_s$, it is true that l_s can be downgraded to l_x via certain computation, which we call an *action*. We use the notation $l_s \overset{a}{\rightsquigarrow} l_x$ to specify the downgrading relation via action a. This is similar to the approach of *intransitive noninterference*, but the key difference is that, the downgrading relation is determined by the semantics of the security levels and the action a performed, and this information is crucial for reasoning about the global effects of downgrading.

3.3 The Road Map

Our framework consists of three parts: *policy specification*, *enforcement mechanism* and the *security guarantee*. The basis of our theory is a well-studied, security-typed language λ_{sec} as shown in Figures 3, 4 and 6, where security levels from the simplest security lattice $\mathbb{L}_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$ are used as type annotations. A *noninterference* theorem can be proved for languages like λ_{sec}.

The rest of this paper is organized in a step-by-step fashion. We first set out to define a lattice of *local* downgrading policies called $\mathbb{L}_{\mathsf{local}}$ in Section 4, where each policy describes how the secret can be downgraded by interacting with constants and low-level public information. Correspondingly, we extend the language λ_{sec} to $\lambda^{\Downarrow}_{\mathsf{local}}$ in Section 5 with typing rules for downgrading. We formalize the security guarantee as a relaxed form of noninterference using program equivalences and prove the soundness of our type system. In Section 6 and 7, we extend $\mathbb{L}_{\mathsf{local}}$ to $\mathbb{L}_{\mathsf{global}}$ with *global* downgrading policies that describes how secrets can be leaked by composing multiple secrets together, and patch the type system to $\lambda^{\Downarrow}_{\mathsf{global}}$ with a similar relaxed noninterference theorem. We discusses the application of this framework in Section 8 and conclude in Section 9.

4. LOCAL DOWNGRADING POLICIES

4.1 Label Definition

Definition 4.1.1 (The policy language). *In Figure 1.*

Types	$\tau ::=$	$\mathsf{int} \mid \tau \to \tau$
Constants	$c ::=$	c_i
Operators	$\oplus ::=$	$+, -, =, \ldots$
Terms	$m ::=$	$\lambda x\!:\!\tau.\ m \mid m\ m \mid x \mid c \mid m \oplus m$
Policies	$n ::=$	$\lambda x\!:\!\mathsf{int}.\ m$
Labels	$l ::=$	$\{n_1, \ldots, n_k\} \quad (k \geq 1)$

Figure 1: $\mathbb{L}_{\mathsf{local}}$ Label Syntax

The core of the policy language is a variant of the simply-typed λ-calculus with a base type, binary operators and constants. A **downgrading policy** is a λ-term that specifies how an integer can be downgraded: when this λ-term is applied to the annotated integer, the result becomes public. A **label** is a non-empty set of downgrading policies, specifying all possible ways to downgrade the data. A label can be an infinite set. Each label represents a security level and can be used as type annotations. For example, if we have $x : \mathsf{int}_{\{m_1\}}$ where m_1 is defined as $\lambda y : \mathsf{int}.\ y\%4$, then the result of the application $(m_1\ x) \equiv x\%4$ is considered a public value. Take the password checking example, we can let $p : \mathsf{int}_{\{m_2\}}$ where m_2 is $\lambda x : \mathsf{int}.\ \lambda y : \mathsf{int}.\ x = y$, so that the application $(m_2\ p) \equiv \lambda y\!:\!\mathsf{int}.\ p = y$ is considered as a public closure, assuring that the only way to leak information about p is to use this closure and perform the comparison with p.

Definition 4.1.2 (Label well-formedness).

1. *A policy language term m is well-typed, iff $\vdash m : \tau$ in the simply-typed λ-calculus.*
2. *A label l is well-formed, iff $\forall n \in l$, n is well-typed.*
3. *Let $\mathbb{L}_{\mathsf{local}}$ be the set of all well-formed labels (both finite and infinite).*

Note. In the rest of the paper, we implicitly assume that all the labels are well-formed in our discussion.

Definition 4.1.3 (Term equivalence). *We use conventional $\beta - \eta$ equivalences for λ-calculus, as defined in Figure 2. We write $m_1 \equiv m_2$ as an abbreviation for $\vdash m_1 \equiv m_2 : \tau$. We write $\Gamma \vdash m_1 \equiv m_2$ as an abbreviation for $\Gamma \vdash m_1 \equiv m_2 : \tau$.*

The rules in Figure 2 are call-by-name equivalences, which may not preserve the termination behavior in a call-by-value semantics. It is important that our policy language has no fixpoints and programs never diverge.

Definition 4.1.4 (Term composition). *If $\vdash m_1 : \tau_1 \to \tau_3$, $\vdash m_2 : \tau_2 \to \tau_1$, then the composition of m_1 and m_2 is defined as: $m_1 \circ m_2 \triangleq \lambda x\!:\!\tau_2.\ m_1\ (m_2\ x)$.*

4.2 Label Interpretation

Each label is syntactically represented as a set of downgrading policies, but the semantics of the label includes more than the specified policies. Generally speaking, if $n \in l$ and $n' \equiv m \circ n$, then n' is also a valid downgrading policy implied by l, because each time we apply n' to the data x annotated by l, it is equivalent to first applying n to x to get a public result $(n\ x)$, then applying m to the public result, so that $m\ (n\ x) \equiv n'\ x$ can also be considered as public. Therefore, a finite label implies infinite number of downgrading policies and we need to define the *interpretation* of a label l to represent *all* downgrading policies such that, when the policy term is applied to the data annotated by l, the result is considered public.

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash m \equiv m : \tau} \quad \textsc{Q-Refl}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau}{\Gamma \vdash m_2 \equiv m_1 : \tau} \quad \textsc{Q-Symm}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau \qquad \Gamma \vdash m_2 \equiv m_3 : \tau}{\Gamma \vdash m_1 \equiv m_3 : \tau} \quad \textsc{Q-Trans}$$

$$\frac{\Gamma, x\!:\!\tau_1 \vdash m_1 \equiv m_2 : \tau_2}{\Gamma \vdash \lambda x\!:\!\tau_1.\ m_1 \equiv \lambda x\!:\!\tau_1.\ m_2 : \tau_1 \to \tau_2} \quad \textsc{Q-Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash m_1 \equiv m_2 : \tau_1 \to \tau_2 \\ \Gamma \vdash m_3 \equiv m_4 : \tau_1\end{array}}{\Gamma \vdash m_1\ m_3 \equiv m_2\ m_4 : \tau_2} \quad \textsc{Q-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash m_1 \equiv m_2 : \mathsf{int} \\ \Gamma \vdash m_3 \equiv m_4 : \mathsf{int}\end{array}}{\Gamma \vdash m_1 \oplus m_3 \equiv m_2 \oplus m_4 : \mathsf{int}} \quad \textsc{Q-BinOp}$$

$$\frac{\Gamma, x\!:\!\tau_1 \vdash m_1 \!:\! \tau_2 \qquad \Gamma \vdash m_2 \!:\! \tau_1}{\Gamma \vdash (\lambda x\!:\!\tau_1.\ m_1)\ m_2 \equiv m_1\{m_2/x\} : \tau_2} \quad \textsc{Q-Beta}$$

$$\frac{\neg\mathrm{free}(x, m) \qquad \Gamma \vdash m \!:\! \tau_1 \to \tau_2}{\Gamma \vdash m \equiv \lambda x\!:\!\tau_1.\ m\ x : \tau_1 \to \tau_2} \quad \textsc{Q-Eta}$$

Figure 2: Term Equivalences $\boxed{\Gamma \vdash m_1 \equiv m_2 : \tau}$

Definition 4.2.1 (Label interpretation).
Let $\mathbb{S}(l)$ denote the semantic interpretation of the label l:

$$\mathbb{S}(l) \triangleq \{n' \mid n' \equiv m \circ n,\ n \in l\}$$

This label semantics enjoys the following properties:

Lemma 4.2.1 (Properties of $\mathbb{S}(l)$).

1. *$l \subseteq \mathbb{S}(l) = \mathbb{S}(\mathbb{S}(l))$.*
2. *$n \in \mathbb{S}(l)$ iff $\exists n' \in l, \exists m,\ n \equiv m \circ n'$.*
3. *$l_1 \subseteq \mathbb{S}(l_2)$ iff $\forall n_1 \in l_1,\ \exists n_2 \in l_2, \exists m,\ n_1 \equiv m \circ n_2$.*
4. *$l_1 \subseteq \mathbb{S}(l_2)$ iff $\mathbb{S}(l_1) \subseteq \mathbb{S}(l_2)$.*

We can now reason about the equivalence of labels with respect to the label semantics. Two labels are considered as structurally equivalent if they denote the same set of downgrading policies:

Definition 4.2.2 (Structural equivalence of labels).
We define the structural equivalence $\equiv_l$ on $\mathbb{L}_{\mathsf{local}}$:

$$l_1 \equiv_l l_2 \quad \textit{iff} \quad \mathbb{S}(l_1) = \mathbb{S}(l_2)$$

Corollary 4.2.1 (Properties of $\equiv_l$).

1. *$l \equiv_l \mathbb{S}(l)$*
2. *$l_1 \equiv_l l_2$ iff $l_1 \subseteq \mathbb{S}(l_2)$ and $l_2 \subseteq \mathbb{S}(l_1)$*

4.3 Label Ordering

To organize $\mathbb{L}_{\mathsf{local}}$ as a lattice, we need to introduce partial ordering among the labels and to define joins and meets.

Definition 4.3.1 (Label ordering).
Let $\sqsubseteq$ be a binary relation on $\mathbb{L}_{\mathsf{local}}$ such that

$$l_1 \sqsubseteq l_2 \quad \textit{iff} \quad \mathbb{S}(l_2) \subseteq \mathbb{S}(l_1)$$

This definition relies on the set inclusion relation of label interpretations. If l_2 has fewer downgrading policies than l_1 has, then l_2 denotes a higher security level. We can allow information to flow from l_1 to l_2 without changing its content. If we use labels as type annotations, the ordering of labels determines the subtyping relation: if $l_1 \sqsubseteq l_2$, then $\mathsf{int}_{l_1} \leq \mathsf{int}_{l_2}$.

Corollary 4.3.1. *$\sqsubseteq$ is a partial order on $\mathbb{L}_{\mathsf{local}}$.*

Corollary 4.3.2. *$l_1 \sqsubseteq l_2$ iff $l_2 \subseteq \mathbb{S}(l_1)$.*

Definition 4.3.2 (Joins and meets).

- *The* **upper bound** *for a set of labels X is a label l such that $x \in X$ implies $x \sqsubseteq l$. The* **join** *or the* **least upper bound** *for X is an upper bound l such that for any other upper bound z of X, it is the case that $l \sqsubseteq z$.*

- *The* **lower bound** *for a set of labels X is a label l such that $x \in X$ implies $l \sqsubseteq x$. The* **meet** *or the* **greatest lower bound** *for X is a lower bound l such that for any other lower bound z of X, it is the case that $z \sqsubseteq l$.*

The notation $\sqcup X$ and $\sqcap X$ denote the join and the meet of X. The notation $l_1 \sqcup l_2$ and $l_1 \sqcap l_2$ denote the join and the meet of $\{l_1, l_2\}$.

Because we defined the partial ordering using subset relation, the joins and meets of labels share the same structure as sets:

Corollary 4.3.3 (Interpreting joins and meets).

1. *$\forall X, \exists l_1, \exists l_2$ such that $l_1 \equiv_l \sqcup X$ and $l_2 \equiv_l \sqcap X$*
2. *$\mathbb{S}(\sqcup X) = \cap(\mathbb{S}(X))$, $\mathbb{S}(l_1 \sqcup l_2) = \mathbb{S}(l_1) \cap \mathbb{S}(l_2)$*
3. *$\mathbb{S}(\sqcap X) = \cup(\mathbb{S}(X))$, $\mathbb{S}(l_1 \sqcap l_2) = \mathbb{S}(l_1) \cup \mathbb{S}(l_2)$*

It is inconvenient to use infinite interpretations to represent the result of join and meets. The following lemma shows how to compute joins and meets directly.

Lemma 4.3.1 (Computing joins and meets).

1. *$l_1 \sqcap l_2 \equiv_l l_1 \cup l_2$.*
2. *$l_1 \sqcup l_2 \equiv_l \{n \mid \exists m_1, \exists m_2, \exists n_1 \in l_1, \exists n_2 \in l_2,$
   $n \equiv m_1 \circ n_1 \equiv m_2 \circ n_2\}$.*

Definition 4.3.3 (Highest and lowest labels).

$$\mathsf{H} \triangleq \sqcup \mathbb{L}_{\mathsf{local}}, \quad \mathsf{L} \triangleq \sqcap \mathbb{L}_{\mathsf{local}}$$

The following lemma shows the beauty of this lattice. H corresponds to the most restrictive downgrading policy, where the secret cannot be leaked by any means. To express such a policy in our policy language, we can use a constant function $\lambda x\!:\!\mathsf{int}.\ c$ as the only policy in the label. The intuition is that this function is completely noninterfering, i.e. we can learn nothing about its input x by studying its output c. On the other hand, L corresponds to the least restrictive policy, where the data itself is already considered as public. The simplest way to express this fact is to use the identity function $\lambda x\!:\!\mathsf{int}.\ x$ as the policy, meaning that we can leak all information about this piece of data.

Lemma 4.3.2. $\mathsf{H} \equiv_l \{\lambda x\!:\!\mathsf{int}.\ c\}$, $\mathsf{L} \equiv_l \{\lambda x\!:\!\mathsf{int}.\ x\}$

Proof:

- For any well-formed label l, we can prove that $(\lambda x : \mathsf{int}.\ c) \in \mathbb{S}(l)$: $\forall n \in l$, suppose $\vdash n : \mathsf{int} \to \tau$, we construct the term $\lambda x : \tau.\ c$ so that $(\lambda x : \tau.\ c) \circ n \equiv_l \lambda x\!:\!\mathsf{int}.\ c$, which implies $(\lambda x\!:\!\mathsf{int}.\ c) \in \mathbb{S}(l)$.

  Therefore, $(\lambda x : \mathsf{int}.\ c) \in \mathbb{S}(\mathsf{H})$, $\{\lambda x : \mathsf{int}.\ c\} \subseteq \mathbb{S}(\mathsf{H})$ and $\mathsf{H} \sqsubseteq \{\lambda x\!:\!\mathsf{int}.\ c\}$. By definition of H we also have $\{\lambda x\!:\!\mathsf{int}.\ c\} \sqsubseteq \mathsf{H}$, so that $\mathsf{H} \equiv_l \{\lambda x\!:\!\mathsf{int}.\ c\}$.

- $\forall n \in \mathsf{L}$, $n \equiv n \circ (\lambda x : \mathsf{int}.\ x)$, so $\mathsf{L} \subseteq \mathbb{S}(\{\lambda x : \mathsf{int}.\ x\})$ and $\{\lambda x\!:\!\mathsf{int}.\ x\} \sqsubseteq \mathsf{L}$. By definition of L, we also have $\mathsf{L} \sqsubseteq \{\lambda x\!:\!\mathsf{int}.\ x\}$, therefore $\mathsf{L} \equiv_l \{\lambda x\!:\!\mathsf{int}.\ x\}$ □

We can further show that all the noninterfering functions are in the interpretation of H. In this particular scenario, constant functions and noninterfering functions have the same meaning. We can also show that all the policy functions, both interfering and noninterfering, are in the interpretation of L. For a label l between H and L, the policy terms precisely define a set of permitted *interfering* functions.

Theorem 4.3.1 (Lattice completeness).
The pair $\langle \mathbb{L}_{\mathsf{local}}, \sqsubseteq \rangle$ is a complete lattice.

4.4 Label Downgrading

Downgrading happens when data is involved in some computation. The security level of data changes depending on the computation performed. We describe such computation as an *action* and formalize downgrading as a ternary relation: $l_1 \overset{a}{\rightsquigarrow} l_2$.

Definition 4.4.1 (Multi-composition).
Suppose $\vdash m_1 : \mathsf{int} \to \tau$, $\vdash m_2 : \tau_1 \to \tau_2 \to \ldots \to \tau_k \to \mathsf{int}$, the multi-composition of m_1 and m_2 is defined as:

$$m_1 \odot m_2 \triangleq \lambda y_1\!:\!\tau_1.\ \ldots \lambda y_k\!:\!\tau_k.\ m_1(m_2\ y_1\ \ldots y_k)$$

Definition 4.4.2 (Actions). *We use the metavariable a to range over actions. An action is a λ-term that has the same syntax as a downgrading policy function. That is, the metavariable a and n range over the same set of terms.*

Definition 4.4.3 (Downgrading relation). *We use the notation $l_1 \overset{a}{\rightsquigarrow} l_2$ to denote that l_1 can be downgraded to l_2 via the action a. Given a well-typed action a, $\overset{a}{\rightsquigarrow}$ is a binary relation on $\mathbb{L}_{\mathsf{local}}$:*

$$l_1 \overset{a}{\rightsquigarrow} l_2 \quad \textit{iff} \quad \forall n_2 \in \mathbb{S}(l_2), n_2 \odot a \in \mathbb{S}(l_1)$$

Example 4.4.1 (Downgrading). *Suppose we have an integer u at security level l_1, where l_1 is defined as:*

$$l_1 \triangleq \{n_1\},\ n_1 \triangleq \lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ \lambda z\!:\!\mathsf{int}.\ (x\%y) = z$$

Suppose we have another integer v at security level L. What is the security level for $(u\%v)$? We can define an action that describes this computation step:

$$a \triangleq \lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ x\%y$$

The result has a security level l_2:

$$l_2 \triangleq \{n_2\},\ n_2 \triangleq \lambda x\!:\!\mathsf{int}.\ \lambda z\!:\!\mathsf{int}.\ x = z$$

It is easy to verify that $l_1 \overset{a}{\rightsquigarrow} l_2$, because $n_2 \odot a \equiv n_1$.

Lemma 4.4.1.

1. *If $l_1 \overset{a}{\rightsquigarrow} l_2$ and $l_2 \sqsubseteq l_3$ then $l_1 \overset{a}{\rightsquigarrow} l_3$.*
2. *If $l_1 \overset{a}{\rightsquigarrow} l_2$ and $l_3 \sqsubseteq l_1$ then $l_3 \overset{a}{\rightsquigarrow} l_2$.*

The above lemma shows very useful properties of downgrading. It implies that if $l_1 \overset{a}{\rightsquigarrow} l_2$, then $l_1 \overset{a}{\rightsquigarrow} \mathsf{H}$, but it is not very useful to use H as the result because it simply forbids any further downgrading. We can see that downgrading is not deterministic: given l_1 and a, there are many targets l_1 that can be downgraded to via a. The questions are: which label is the most useful result, and how to find it?

Definition 4.4.4 (Lowest downgrading). *Let $\Downarrow (l, a)$ be the greatest lower bound of all possible labels that l can be downgraded to via a:*

$$\Downarrow (l, a) \triangleq \sqcap\{l' \mid l \overset{a}{\rightsquigarrow} l'\}$$

Lemma 4.4.2.

1. *$l \overset{a}{\rightsquigarrow} \Downarrow (l, a)$*
2. *If $l \overset{a}{\rightsquigarrow} l'$ then $\Downarrow (l, a) \sqsubseteq l'$*

The above lemma shows that $\Downarrow (l, a)$ is the most accurate (lowest) label that l can be downgraded to via a. In fact, given l and a, all the labels l' that satisfy $l \overset{a}{\rightsquigarrow} l'$ form a sublattice of $\mathbb{L}_{\mathsf{local}}$, where the bottom of the lattice is $\Downarrow (l, a)$ and the top is H.

Lemma 4.4.3 (Computing downgrading results).

$$\Downarrow (l, a) \equiv_l \{n \mid \exists n_1 \in l, \exists m, n \odot a \equiv m \circ n_1\}$$

This lemma shows exactly what is inside $\Downarrow (l, a)$.

5. A TYPE SYSTEM FOR LOCAL DOWNGRADING

5.1 The Language

In this section we present a security-typed programming language $\lambda^{\Downarrow}_{\mathsf{local}}$ that supports downgrading. The language syntax is presented in Figure 3. Compared to the policy language we presented in the last section, we introduce conditionals and fixpoints. Security labels are used as type annotations. Furthermore, the inputs to the program are explicitly written as variables: σ denotes a secret input and ω denotes a public input.

Labeled	$s ::=$	t_l
types	$t ::=$	$\mathsf{int} \mid (s \to s)$
Programs	$e ::=$	$(\lambda x\!:\!s.\ e)_l \mid e\ e \mid x \mid c \mid \sigma \mid \omega$
		$\mid e \oplus e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid \mathsf{fix}_l\ r(x) = e \mid r$
Secret inputs	$\sigma ::=$	σ_i
Public inputs	$\omega ::=$	ω_i

Figure 3: λ_{sec}, $\lambda^{\Downarrow}_{\mathsf{local}}$ Syntax

Definition 5.1.1 (Local Downgrading Policies). *Let $\Sigma(\sigma_i)$ denote the security label for σ_i.*

In this system, we aim for an end-to-end style security guarantee. For each secret input σ_i of the program, the user specifies a label $\Sigma(\sigma_i)$ as its downgrading policy. For example, the policy for the password may be:

$$\Sigma(\sigma_{pwd}) = \{(\lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ x = y\}$$

which only allows downgrading by comparing the password to a value at security level L. The policy for the variable `secret` in Example 3.2.1 can be written as:

$$\Sigma(\sigma_{secret}) = \{(\lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ (\mathrm{hash}(x)\%2^{64}) = y\}$$

where the hash function is a function provided by the external library and it can be modeled as an operator in our system.

5.2 The Type System

Definition 5.2.1 (Type stamping). $t_{l_1} \sqcup l_2 \triangleq t_{(l_1 \sqcup l_2)}$.

Most common typing rules are in Figure 4 and we call them λ_{sec} rules, because they are standard typing rules in traditional security-typed languages. The downgrading rule is in Figure 5. We only listed the DLocal-Left rule, and omitted it symmetrical case, the DLocal-Right rule. The subtyping rules are listed in Figure 6.

For simplicity, we require that all the fixpoint functions have type $(\mathsf{int}_l \to \mathsf{int}_l)_l$. As a design choice, we do not allow loop variables have security levels other than L and H. The reason is that a loop variable changes its own values during recursive calls. In our security lattice, the security level of data downgrades during computation unless it is L or H. Since all the policy terms are terminating programs, the security level of data always becomes L or H after finite steps of nontrivial computation.

5.3 The Security Goal

If we erase the type annotations, the unlabeled programs in Figure 7 is a superset of our policy language in Figure 1, so that we can use terms in our policy language to represent fragments of unlabeled programs.

Definition 5.3.1 (Label erasure). *$\mathcal{E}(e)$ erases all the label annotations in e and returns a simply-typed λ-term, as defined in Figure 7.*

Definition 5.3.2 (Term sanity). *The predicate $\mathsf{clean}(f)$ holds if and only if f syntactically contains no secret variable σ.*

Definition 5.3.3 (Program equivalences). *All the rules in Figure 2 are also used for program equivalences by substituting all metavariables m with f. Furthermore, we have some new rules defined in Figure 8.*

$$\Gamma \vdash c_i : \mathsf{int}_{\mathsf{L}} \qquad \textsc{TConst}$$

$$\Gamma \vdash \omega_i : \mathsf{int}_{\mathsf{L}} \qquad \textsc{TPublic}$$

$$\Gamma \vdash \sigma_i : \mathsf{int}_{\Sigma(\sigma_i)} \qquad \textsc{TSecret}$$

$$\frac{\Gamma(x) = s}{\Gamma \vdash x : s} \qquad \textsc{TVar}$$

$$\frac{\Gamma(r) = s}{\Gamma \vdash r : s} \qquad \textsc{TRecVar}$$

$$\frac{\Gamma, x : s_1 \vdash e : s_2 \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x{:}s_1.\ e)_l : (s_1 \to s_2)_l} \qquad \textsc{TFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (s_1 \to s_3)_l \\ \Gamma \vdash e_2 : s_2 \qquad s_2 \leq s_1 \qquad s_3 \sqcup l \leq s\end{array}}{\Gamma \vdash e_1\ e_2 : s} \qquad \textsc{TApp}$$

$$\frac{\begin{array}{c}l \in \{\mathsf{L}, \mathsf{H}\} \qquad s \leq \mathsf{int}_l \\ \Gamma, r : (\mathsf{int}_l \to \mathsf{int}_l)_l, x : \mathsf{int}_l \vdash e : s\end{array}}{\Gamma \vdash \mathsf{fix}_l\ r(x) = e : (\mathsf{int}_l \to \mathsf{int}_l)_l} \qquad \textsc{TFix}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{int}_l \qquad \Gamma \vdash e_1 : s_1 \\ \Gamma \vdash e_2 : s_2 \qquad s_1 \leq s \qquad s_2 \leq s\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : s \sqcup \mathsf{H}} \qquad \textsc{TCond}\text{-}\mathsf{H}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{int}_{\mathsf{L}} \qquad \Gamma \vdash e_1 : s_1 \\ \Gamma \vdash e_2 : s_2 \qquad s_1 \leq s \qquad s_2 \leq s\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : s} \qquad \textsc{TCond}\text{-}\mathsf{L}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int}_{l_1} \qquad \Gamma \vdash e_2 : \mathsf{int}_{l_2}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}_{\mathsf{H}}} \qquad \textsc{TOp}\text{-}\mathsf{H}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int}_{\mathsf{L}} \qquad \Gamma \vdash e_2 : \mathsf{int}_{\mathsf{L}}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}_{\mathsf{L}}} \qquad \textsc{TOp}\text{-}\mathsf{L}$$

Figure 4: λ_{sec} Typing Rules: $\boxed{\Gamma \vdash e : s}$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{int}_{l_1} \qquad \Gamma \vdash e_2 : \mathsf{int}_{\mathsf{L}} \\ a \triangleq \lambda x{:}\mathsf{int}.\ \lambda y{:}\mathsf{int}.\ x \oplus y \qquad l_1 \overset{a}{\rightsquigarrow} l_3\end{array}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}_{l_3}} \qquad \textsc{DLocal-L(R)}$$

Figure 5: $\lambda^{\Downarrow}_{\mathsf{local}}$ Typing Rules: $\boxed{\Gamma \vdash e : s}$

$$\frac{l_1 \sqsubseteq l_2}{\vdash t_{l_1} \leq t_{l_2}} \qquad \textsc{SLab}$$

$$\vdash t \leq t \qquad \textsc{SRefl}$$

$$\frac{\vdash t_1 \leq t_2 \qquad \vdash t_2 \leq t_3}{\vdash t_1 \leq t_3} \qquad \textsc{STrans}$$

$$\frac{\vdash s_1 \leq s_2 \qquad \vdash s_3 \leq s_4}{\vdash s_2 \to s_3 \leq s_1 \to s_4} \qquad \textsc{SFun}$$

Figure 6: λ_{sec}, $\lambda^{\Downarrow}_{\mathsf{local}}$ Subtyping Rules: $\boxed{\vdash s \leq s}$ $\boxed{\vdash t \leq t}$

Unlabeled Programs

$$\begin{array}{lcl}
f & ::= & \lambda x{:}\tau.\ f \mid f\ f \mid x \mid c \mid \omega \mid \sigma \\
 & & \mid f \oplus f \mid \mathsf{if}\ f\ \mathsf{then}\ f\ \mathsf{else}\ f \\
 & & \mid \mathsf{fix}\ r(x) = f \mid r
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E} : e \to f & & \\
\mathcal{E}(t_l) & = & \mathcal{E}(t) \\
\mathcal{E}(\mathsf{int}) & = & \mathsf{int} \\
\mathcal{E}(s \to s) & = & \mathcal{E}(s) \to \mathcal{E}(s) \\
\mathcal{E}(\Gamma)(x) & = & \mathcal{E}(\Gamma(x)) \\
\mathcal{E}((\lambda x{:}s.\ e)_l) & = & \lambda x{:}\mathcal{E}(s).\ \mathcal{E}(e) \\
\mathcal{E}(e_1\ e_2) & = & \mathcal{E}(e_1)\ \mathcal{E}(e_2) \\
\mathcal{E}(x \mid c \mid \sigma \mid \omega \mid r) & = & x \mid c \mid \sigma \mid \omega \mid r \\
\mathcal{E}(e_1 \oplus e_2) & = & \mathcal{E}(e_1) \oplus \mathcal{E}(e_2) \\
\mathcal{E}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \mathsf{if}\ \mathcal{E}(e_1)\ \mathsf{then}\ \mathcal{E}(e_2)\ \mathsf{else}\ \mathcal{E}(e_3) \\
\mathcal{E}(\mathsf{fix}_l\ r(x) = e) & = & \mathsf{fix}\ r(x) = \mathcal{E}(e)
\end{array}$$

Figure 7: Label Erasure

We formalize the security guarantee of our type system using program equivalences. The following is the main theorem of this paper.

Theorem 5.3.1 (Relaxed noninterference).
If $\vdash e : \mathsf{int}_{\mathsf{L}}$, *then* $\mathcal{E}(e) \equiv f\ (n_1\sigma_{i_1}) \ldots (n_k\sigma_{i_k})$ *where* $\mathsf{clean}(f)$ *and* $\forall j. n_j \in \Sigma(\sigma_{i_j})$.

The proof of this theorem is in Subsection 5.5. This theorem shows that a type-safe program can only leak secret information in controlled ways, i.e. only through the specified downgrading functions. Take the password example again, if we know that

$$\mathcal{E}(e) \equiv f\ ((\lambda x{:}\mathsf{int}.\ \lambda y{:}\mathsf{int}.\ x = y)\ \sigma_{pwd})$$

and $\mathsf{clean}(f)$, then the only way through which f can leak information about σ_{pwd} is to use its argument, the closure $(\lambda y : \mathsf{int}.\ \sigma_{pwd} = y)$, which intuitively enforces the security policy specified by the user in an end-to-end fashion. Note that this policy still allows the full password be leaked by the following program:

$f \triangleq \lambda g{:}\mathsf{int} \to \mathsf{int}.\ (\mathsf{fix}\ r(x) = \mathsf{if}\ g(x)\ \mathsf{then}\ x\ \mathsf{else}\ r(x+1))\ 0$

Nevertheless, such an attack takes exponentially long time to finish. We will discuss such programs more in Section 8.

We call this security guarantee *relaxed noninterference*, because it generalizes traditional noninterference as shown in the following corollary.

All the $\Gamma \vdash m_1 \equiv m_2 : \tau$ rules become $\Gamma \vdash f_1 \equiv f_2 : \tau$, plus the following rules:

$$\frac{\begin{array}{c}\Gamma \vdash f_1 \equiv f_2 : \mathsf{int} \\ \Gamma \vdash f_3 \equiv f_4 : \tau \qquad \Gamma \vdash f_5 \equiv f_6 : \tau\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{if}\ f_1\ \mathsf{then}\ f_3\ \mathsf{else}\ f_5 \\ \equiv \mathsf{if}\ f_2\ \mathsf{then}\ f_4\ \mathsf{else}\ f_6 : \tau\end{array}} \quad \text{Q-IF}$$

$$\frac{\Gamma, x : \mathsf{int}, r : \mathsf{int} \to \mathsf{int} \vdash f_1 \equiv f_2 : \mathsf{int}}{\Gamma \vdash \mathsf{fix}\ r(x) = f_1 \equiv \mathsf{fix}\ r(x) = f_2 : \mathsf{int}} \quad \text{Q-FIX}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{if}\ f_1\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3 : \tau_1 \to \tau_2 \\ \Gamma \vdash f_4 : \tau_1\end{array}}{\begin{array}{c}\Gamma \vdash (\mathsf{if}\ f_1\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3)\ f_4 \\ \equiv \mathsf{if}\ f_1\ \mathsf{then}\ f_2\ f_4\ \mathsf{else}\ f_3\ f_4 : \tau_2\end{array}} \quad \text{Q-ETAIF-APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{if}\ f_1\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3 : \mathsf{int} \\ \Gamma \vdash f_4 : \mathsf{int}\end{array}}{\begin{array}{c}\Gamma \vdash (\mathsf{if}\ f_1\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3) \oplus f_4 \\ \equiv \mathsf{if}\ f_1\ \mathsf{then}\ f_2 \oplus f_4\ \mathsf{else}\ f_3 \oplus f_4 : \mathsf{int}\end{array}} \quad \text{Q-ETAIF-OP-L(R)}$$

Figure 8: Program Equivalences: $\boxed{\Gamma \vdash f_1 \equiv f_2 : \tau}$

Corollary 5.3.1 (Pure noninterference).
If $\vdash e : \mathsf{int}_\mathsf{L}$ *and* $\forall j.\Sigma(\sigma_j) = \mathsf{H}$*, then* $\mathcal{E}(e) \equiv f$ *where* $\mathsf{clean}(f)$.

Obviously, when no downgrading policy is available, a type-safe program is noninterfering because it is equivalent to another program that contains no secret variable at all, which implies that the program does not leak any information about the secret variables.

It is important to understand the meaning of the equivalence rules. We treat these rules as the static semantics of the program. Rather than evaluating the program in a call-by-value semantics, we transform the program statically in a call-by-name fashion and formalize our security goal. In a call-by-value setting, the Q-BETA and Q-ETA rules affect the termination behavior of the program. The Q-ETAIF rules allow us to statically reason about different execution paths without changing the termination behavior of the program. If $f_1 \equiv f_2$ and both programs terminate in a call-by-value semantics, they must evaluate to the same value. With such equivalence rules, our relaxed noninterference theorem gives us a notion of *weak noninterference*, where secrets can be leaked by observing the termination behavior of program.

5.4 Making Typechecking Practical

During typechecking, we need tractable ways to work with the security labels. The major label operations in the typing rules are: order testing, computing joins and computing downgrading results. There are two challenges here. First, some label operations involve higher-order unification problems that require searching and such problems are undecidable. Second, labels with infinite size are hard to deal with.

Although higher-order unification is generally undecidable, most such problems in typechecking are either trivial or easily solvable. Take the label ordering as an example, we can use the following corollary to test whether $l_1 \sqsubseteq l_2$:

Corollary 5.4.1 (Label order testing).

1. *If* $l_1 \subseteq l_2$ *then* $l_2 \sqsubseteq l_1$.
2. $l_2 \sqsubseteq l_1$ *iff* $\forall n_1 \in l_1, \exists n_2 \in l_2, \exists m,\ n_1 \equiv m \circ n_2$

In typechecking, it is often the case that one of l_1 and l_2 are either H or L, or $l_1 \subseteq l_2$. It is rarely the case that we need to search for the unifier m, and if we need to do so, the size of m is usually no larger than n_1, because the computation of n_1 is being decomposed into two steps, and each piece is likely to have fewer computation than n_1 does. If no unifier is found within the length of n_1, the typechecker could conservatively report that the label ordering cannot be established, as doing so does not break type soundness.

We solve the finite representation problem by *approximating* intractable labels. Suppose we do not know how to represent l finitely and for some policies we cannot even decide whether they are in $\mathbb{S}(l)$. But, if we can compute a finite label l' such that we know $l' \subseteq \mathbb{S}(l)$, then we have $l \sqsubseteq l'$ and l' can be used as an approximation for l. To make such approximations useful, l' should be as close to l as possible.

The following shows how to approximate joins and downgrading results for finite labels.

Lemma 5.4.1 (Approximating joins).
$l_1 \sqcup l_2 \sqsubseteq l$ *where* $l \triangleq \{\lambda x : \mathsf{int}.\ c\} \cup \{n \mid n \in l_1$ *and* $n \in \mathbb{S}(l_2)\} \cup \{n \mid n \in l_2$ *and* $n \in \mathbb{S}(l_1)\}$

In most cases, we have either $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$ and the computation of joins can be short-circuited. In some rare cases, the join of l_1 and l_2 can be approximated by using the above lemma: for each policy $n \in l_1$, test whether it is implied by l_2 and vice versa. The member test $n \in \mathbb{S}(l_2)$ uses Lemma 4.2.1 and unification can be handled as we just did for label ordering.

The TAPP and the TCOND rules do not require the exact join to be computed, so this approximation can always be used.

Lemma 5.4.2 (Approximating downgrading results).
$\Downarrow (l, a) \sqsubseteq \{n \mid \exists n_1 \in l, n \odot a \equiv n_1\}$

This lemma can be used to optimize the searching in Lemma 4.4.3. The intuition is that, a is usually a minimal step of computation in our type system and n_1 is usually a long sequence of computation that can be decomposed into smaller steps.

Therefore, we have a practical procedure for finding the approximation of $\Downarrow (l, a)$: for each $n_1 \in l$, we search for n such that $n \odot a \equiv n_1$. Since g is usually a policy shorter than n_1, most sensible answers can be found by searching for terms no larger than n_1.

By Lemma 4.4.1, the approximated result can be safely used in typechecking.

5.5 Proof of Theorem 5.3.1

This proof involves two stages. First, we transform the program into a normal form defined in Definition 5.5.1. The transformation takes finite steps and preserves program equivalences. Then, we use induction to prove the theorem for normalized programs.

Definition 5.5.1 (Normal forms). *In Figure 9.*

$$
\begin{aligned}
v ::= \quad & x \mid c \mid \sigma \mid \omega \mid v \oplus v \mid \textsf{if } v \textsf{ then } v \textsf{ else } v \\
& \mid (\textsf{fix}_l\ r(x) = v)\ v \mid r\ v
\end{aligned}
$$

Figure 9: Normal forms

The key idea about normal forms is that the metavariable v always ranges over terms of the $\textsf{int}$ type. To transform a program into a normal form, we would like to use β-reductions to get rid of all the λ-abstractions. The exception is that the left side of an application node may not be a λ-abstraction: it can be a fixpoint, a variable or a branch. In Definition 5.5.2, we also defined $\eta_{\textsf{if}}$-reduction. Lemma 5.5.2, Lemma 5.5.3 and Lemma 5.5.4 tell us that these two reduction rules are sufficient to normalize a well-typed program.

Definition 5.5.2 ($\beta, \eta_{\textsf{if}}$ reductions). *In Figure 10.*

Lemma 5.5.1 (Equivalence preservation).
If $e \Rightarrow^* e'$, then $\mathcal{E}(e) \equiv \mathcal{E}(e')$.

Lemma 5.5.2 (Progress under $\beta, \eta_{\textsf{if}}$ reductions).
If $\vdash e : \textsf{int}_l$, e is stuck under $\beta, \eta_{\textsf{if}}$ reduction, then $e = v$ as in the normal form defined in Figure 9.

Lemma 5.5.3 (Preservation under $\beta, \eta_{\textsf{if}}$ reductions).
If $\vdash e : s$, $e \Rightarrow e'$, then $\vdash e' : s'$ where $s' \leq s$.

Lemma 5.5.4 (Normalization under $\beta, \eta_{\textsf{if}}$ reductions).
If $\vdash e : \textsf{int}_l$, then $\exists v$ such that $e \Rightarrow^* v$.

$$
\begin{aligned}
\mathbb{E} ::= \quad & [\,] \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathbb{E} \oplus e \mid v \oplus \mathbb{E} \\
& \mid \textsf{if } \mathbb{E} \textsf{ then } e \textsf{ else } e \ \mid \textsf{if } v \textsf{ then } \mathbb{E} \textsf{ else } e \\
& \mid \textsf{if } v \textsf{ then } v \textsf{ else } \mathbb{E} \ \mid \textsf{fix}_l\ r(x) = \mathbb{E}
\end{aligned}
$$

$$
\begin{aligned}
& \Rightarrow_{\beta,\eta_{\textsf{if}}} : e \rightarrow e \\
& \mathbb{E}[(\lambda x{:}s.\ e_1)\ e_2] \Rightarrow_\beta \mathbb{E}[e_1\{e_2/x\}] \\
& \mathbb{E}[(\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3)\ e_4] \Rightarrow_{\eta_{\textsf{if}}} \mathbb{E}[\textsf{if } e_1 \textsf{ then } e_2\ e_4 \textsf{ else } e_3\ e_4]
\end{aligned}
$$

Figure 10: β and $\eta_{\textsf{if}}$ reduction rules

$$
\begin{aligned}
& \textsf{branch}(F_C, F) ::= \qquad\qquad (f_i \in F,\ f_{c_j} \in F_C) \\
& f_i \mid \textsf{if } f_{c_j} \textsf{ then branch}(F_C, F) \textsf{ else branch}(F_C, F)
\end{aligned}
$$

Figure 11: Short hand notion for nested branches

Definition 5.5.3 (Branches). *In Figure 11.*

Definition 5.5.3 a short hand notion for representing nested branching statements. It is easy to show that

$$\Gamma \vdash \textsf{branch}(F_C, F) \oplus f \equiv \textsf{branch}(F_C, F \oplus f)$$

and vice versa. The proof of our main lemma proceeds with induction on the typing derivation of a normalized program.

Lemma 5.5.5 (Main lemma). *Suppose Γ only contains variables introduced by the* TFIX *rule. That is, $\Gamma \vdash r : (\textsf{int}_l \rightarrow \textsf{int}_l)$, $\Gamma \vdash x : \textsf{int}_l$, $l \in \{\textsf{L}, \textsf{H}\}$ for all r and x in Γ.*

1. *If $\Gamma \vdash v : \textsf{int}_\textsf{L}$, then $\mathcal{E}(\Gamma) \vdash \mathcal{E}(v) \equiv f\ (n_1\sigma_{i_1}) \ldots (n_k\sigma_{i_k})$ where $\textsf{clean}(f)$ and $\forall j. n_j \in \Sigma(\sigma_{i_j})$.*

2. *If $\Gamma \vdash v : \textsf{int}_l$, $l \neq \textsf{H}$, $l \neq \textsf{L}$, then*

   (a) $\mathcal{E}(\Gamma) \vdash \mathcal{E}(v) \equiv \textsf{branch}(F_C, F)$
   where $F_C = \{\mathcal{E}(v_{01}), \ldots, \mathcal{E}(v_{0k})\}$,
   $F = \{(a_1\ \sigma_{a_1})\ \mathcal{E}(v_{11}) \ldots \mathcal{E}(v_{1k_1}),$
   $\ldots\ldots\ldots$
   $(a_j\ \sigma_{a_j})\ \mathcal{E}(v_{j1}) \ldots \mathcal{E}(v_{jk_j})\}$

   (b) $\Gamma \vdash v_{ij} : \textsf{int}_\textsf{L}$, *and the typing derivation is smaller than* $\Gamma \vdash v : \textsf{int}_l$

   (c) $\Sigma(\sigma_{a_i}) \overset{a_i}{\leadsto} l$ *for all i.*

Proof: By induction on $\Gamma \vdash v : \textsf{int}_l$.

- Case TCONST, TPUBLIC : The type must be $\textsf{int}_\textsf{L}$. Simply let f be v.
- Case TSECRET : Choose the secret variable itself σ_i, let $a_1 = \lambda x{:}\textsf{int}.\ x$.
- Case TFUN,TRECVAR : Cannot happen.
- Case TVAR : By our assumption on Γ, x must have type $\textsf{int}_\textsf{L}$. Same as the TCONST case.
- Case TAPP : v is either $(\textsf{fix}_l\ r(x) = v_1)\ v_2$ or $r\ v_2$. For the fix subcase, we must have $l = \textsf{L}$, otherwise v will have type $\textsf{int}_\textsf{H}$. For the r subcase, we know from our assumption about Γ that r have type $\textsf{int}_\textsf{L} \rightarrow \textsf{int}_\textsf{L}$. By inversion we know that the type of v_2 must be a subtype of $\textsf{int}_\textsf{L}$, which implies that v_2 must have type $\textsf{int}_\textsf{L}$ in the premises of TAPP. So we can use IH(1) on v_2 and get $\mathcal{E}(\Gamma) \vdash \mathcal{E}(v_2) \equiv f_2$....

  For the fix subcase, we can extend Γ with r and x and use our IH(1) to go into v_1 and get $\mathcal{E}(\Gamma, r, x) \vdash \mathcal{E}(v_1) \equiv f_1$.... Use the QFIX Rule, we have $\mathcal{E}(\Gamma) \vdash \mathcal{E}(\textsf{fix}_l\ r(x) = v_1) \equiv \textsf{fix}\ r(x) = f_1$... Then we can compose f_1 and f_2 to prove (1). The other subcase $r\ v_2$ is similar.
- Case TFIX, TOP-H, TCOND-H : Cannot happen.
- Case TOP-L : Use IH(1) and equivalence rules.
- Case TCOND-L : First we can use IH(1) on e. Then we assert that both branches have $\textsf{int}$ type.

  If $s = \textsf{int}_\textsf{L}$, then we know that both s_1 and s_2 are $\textsf{int}_\textsf{L}$, so that we can use IH(1) and simple equivalence rules to prove this case.

  If $s \neq \textsf{int}_\textsf{L}$, then we use IH(2) on e_1 and e_2 respectively, then compose the result. The downgrading condition in (2)(c) is preserved by some property of the downgrading relation.
- Case DLOCAL-LEFT : If l_1 is $\textsf{H}$ then we can show that it is impossible. If $l_1 = \textsf{L}$ and $l_3 = \textsf{L}$ then we can use IH(1) to prove (1). If $l_1 = \textsf{L}$ and $l_3 \neq \textsf{L}$ then we can create a vacuous secret and put a constant function to prove (2).

  Consider the subcase when $l_1 \neq \textsf{L}$ and $l_1 \neq \textsf{H}$. Use IH(2) to prove (2a),(2b),(2c) ... and do a case analysis on the resulting label l_3. If $l_3 \neq \textsf{L}$ then we proved (2), otherwise use IH again to prove (1).
- Case DLOCAL-RIGHT : Similar.

□

Finally, we can easily compose Lemma 5.5.4, Lemma 5.5.1 and Lemma 5.5.5 to prove Theorem 5.3.1.

6. GLOBAL DOWNGRADING POLICIES

6.1 Motivation

In the last two sections we presented a system with *local* downgrading, where each secret is assigned a security label and secrets can be downgraded by interacting with public inputs and constants. In practice, this framework is capable of expressing many useful downgrading policies, but there are some important policies it cannot express. For example, we may want to specify the policy "data must be encrypted before sending it to the network". Naively we can use the policy $\lambda x:\mathsf{int}.\ \mathrm{encrypt}(x)$ and treat "encrypt" as an operator in our framework. However, an encryption algorithm usually requires a key as its input, so we may try the policy $\lambda x:\mathsf{int}.\ \lambda y:\mathsf{int}.\ \mathrm{encrypt}(x,y)$ for the data and $\lambda x:\mathsf{int}.\ \lambda y:\mathsf{int}.\ \mathrm{encrypt}(y,x)$ for the key. Unfortunately, this does not work because the downgrading rule requires the secrets interact with an $\mathsf{int_L}$ type. Furthermore, these policies allow the attacker to use its own key to downgrade the secret: $\mathrm{encrypt}(x,\mathrm{fakekey})$.

Another interesting example is: we have two secrets σ_1 and σ_2 and we want to specify the policy "both σ_1 and σ_2 are secrets, but their sum is considered as public". Such policies not only describe the computation required for downgrading, but also specifies how multiple secrets should be composed in order to downgrade.

We solve this problem by introducing the idea of *global* downgrading policies. We identify all the secret inputs of the system, and refer to these secrets in our policy language. In this section we present $\mathbb{L}_{\mathsf{global}}$, a lattice of global downgrading policies, and in the next section we correspondingly extend the type system to support global downgrading.

6.2 Label Definition

The only thing we need to change in the policy language is to allow secret variables to appear in the policy language, as shown in Figure 12. For example, σ_1 may have a downgrading policy $\{\lambda x:\mathsf{int}.\ x+\sigma_2\}$, and when we apply this policy term to σ_1, the resulting term $\sigma_1+\sigma_2$ is considered public. Similarly, σ_2 can have the policy $\{\lambda x:\mathsf{int}.\ \sigma_1+x\}$. We use $\mathbb{L}_{\mathsf{global}}$ to denote the set of all well-formed labels.

Policy Terms $\quad m ::= \quad ... \mid \sigma$

Figure 12: $\mathbb{L}_{\mathsf{global}}$ Label Syntax

6.3 Label Interpretation

The label interpretation is slightly different from $\mathbb{L}_{\mathsf{local}}$. The general idea remains the same. If $n \in l$, then $m \circ n$ is implied by n. However, we must assure that m does not contain other secrets, otherwise by applying $m \circ n$ to the data, we may leak arbitrary secrets by deliberately choosing some m. Therefore, we need to make a patch to our definition.

Definition 6.3.1 (Label Interpretation). *Let $\mathbb{S}(l)$ denote the semantic interpretation of the label l:*

$$\mathbb{S}(l) = \{n' \mid n' \equiv m \circ n,\ n \in l, \mathsf{clean}(m)\}$$

Lemma 4.2.1 requires a similar patch. Others parts require no change in Subsection 4.2.

6.4 Label Ordering

The definition of label ordering in Subsection 4.3 requires no change. Lemma 4.3.1 requires a similar patch as above. The interesting thing is that Lemma 4.3.2, which asserts that the identity function is the bottom of the lattice, becomes broken. For backward compatibility, we change our definition for the rest of the paper:

Definition 6.4.1. $\mathsf{H} \triangleq \{\lambda x:\mathsf{int}.\ c\}$, $\mathsf{L} \triangleq \{\lambda x:\mathsf{int}.\ x\}$

It is easy to verify that $\mathsf{H} \equiv_l \sqcup\mathbb{L}_{\mathsf{global}}$ still holds, but $\sqcap\mathbb{L}_{\mathsf{global}}$ is no longer structurally equivalent to L. The intuition is that a constant function is still the most restrictive policy because it leaks no information. The identity function is no longer the least restrictive policy: it can only leak information about the data it annotates. But there are plenty of policies that allow leakage of information besides the annotated data itself. Take this policy as an example: $\lambda x:\mathsf{int}.\ \lambda y:\mathsf{int}.\ x*(y=0)+\sigma_1*(y=1)$, it is capable of leaking the annotated data as well as another secret σ_1. Intuitively, we can try to quantify the information leakage of policies: constant functions leak 0 unit of information, identity functions leak 1 unit, all policies in $\mathbb{L}_{\mathsf{local}}$ leak between 0 and 1, and some policies in $\mathbb{L}_{\mathsf{global}}$ leak much more than 1.

It turns out that if we add tuples and projections in our policy language and enrich the equivalence rules, we can easily give a simple finite representation of $\sqcap\mathbb{L}_{\mathsf{global}}$, which we call Bottom. Assuming the secret variables in the system are $\sigma_1, ..., \sigma_k$, then

$$\mathsf{Bottom} \triangleq \sqcap\mathbb{L}_{\mathsf{global}} \equiv_l \{\lambda x:\mathsf{int}.\ \langle x, \sigma_1, ..., \sigma_k\rangle\}$$

Such a function is capable of leaking all possible secrets besides the annotated data itself.

Although adding Bottom helps us understand the structure of $\mathbb{L}_{\mathsf{global}}$, we do not need it in practice. The security level L still has important practical meaning: if x is annotated with a label l and we have $l \sqsubseteq \mathsf{L}$, then x can still be considered as public. It is only different when x has the ability to interact with other secrets and downgrade them.

6.5 Downgrading

All the definitions in Subsection 4.4 require no modification, except that we need the unifiers m to be clean in Lemma 4.4.3. The actions now can contain secret variables. For example, we have

$$\{\lambda x:\mathsf{int}.\ (x+\sigma_2)\%4\} \overset{a}{\rightsquigarrow} \{\lambda x:\mathsf{int}.\ x\%4\}$$

where $a \triangleq \lambda y:\mathsf{int}.\ y+\sigma_2$. In fact, the secret variables are handled just like constants.

7. A TYPE SYSTEM FOR GLOBAL DOWNGRADING

7.1 Integrity Labels

In this section, we extend the $\mathbb{L}_{\mathsf{local}}$ language in order to support global downgrading policies. As we add the secret variables in the downgrading policy, there are some new issues to solve. Consider the simplest case where we are going to typecheck a term $a+b$. Suppose we already know that a has a security level $\{\lambda x:\mathsf{int}.\ x+\sigma_2\}$. We define an action $\lambda y:\mathsf{int}.\ y+b$ and attempt to downgrade a via this action so that the result can have security level L. In order to do that, it is necessary to establish that the term b must be equal to σ_2. More generally speaking, we need some integrity reasoning about the data, and it is the dual of the confidentiality analysis we have done. The downgrading policies mainly express confidentiality requirements: where the data can go to and what kind of computation we must do before releasing it to the public. To enforce such policies, we also need integrity analysis of data: where the data comes from and what computation has been done with them.

Since integrity and confidentiality are duals, it is natural to use a dual mechanism to reason about integrity. We introduce an optional type annotation, called an integrity label in our language. Such labels can be attached to the base type in the form of $\mathsf{int}\langle m\rangle$ as in Figure 13, where m tracks the interesting computations that happened to this term. For example, a term of type $\mathsf{int}\langle\sigma_1\rangle_l$ must be equivalent to σ_1 itself and this is just a singleton type; a term of type $\mathsf{int}\langle\lambda x\!:\!\mathsf{int}.\ x*\sigma_2\rangle_l$ must be equivalent to $y*\sigma_2$ where y is another term of type int_L. The integrity labels are essentially the dual of our confidentiality labels. The difference is that the integrity label is optional and it has exactly one policy term in it.

$$\begin{array}{lrl}
\text{Labeled Types} & s ::= & t_l \\
 & t ::= & \mathsf{int} \mid \mathsf{int}\langle m\rangle \mid (s \rightarrow s) \\
\text{Global Policies} & \Sigma ::= & \{m_i\} \cup \{\mathsf{H}\}
\end{array}$$

Figure 13: $\lambda^{\Downarrow}_{\mathsf{global}}$ Syntax

7.2 Policy Splitting

If we directly specify the downgrading policy for each secret input just as we did for $\lambda^{\Downarrow}_{\mathsf{local}}$, we are likely to have some inconsistencies among these policies. Take the example of $\sigma_1+\sigma_2$ again. If the downgrading policy for σ_1 is $\{\lambda x:\mathsf{int}.\ x+\sigma_2\}$ and the policy for σ_2 is just H, can we downgrade $\sigma_1+\sigma_2$ to L? The policy of σ_1 says yes and the policy of σ_2 says no. To be safe, we have to compute the downgrading result from both sides, and take the upper bound of them. Doing so will produce a result of H, which is absolutely safe but inconvenient. If the user actually wants such downgrading to be successful, he or she has to write a symmetric policy for σ_2. Such work is tedious and error-prone when the policies become complicated.

To guarantee the consistency of such policies, we change the method of policy specification. Instead of writing policies for individual secrets, the user simply writes a set Σ of policy terms as shown in Figure 13. Each of these terms in Σ denotes a way of downgrading secrets to public. For example, we can have $\Sigma=\{m_1,m_2,m_3,\mathsf{H}\}$ where

- $m_1 \triangleq (\sigma_1\%2)$, meaning that σ_1 can be downgraded to public by exposing its parity;
- $m_2 \triangleq (\lambda x\!:\!\mathsf{int}.\ \sigma_2=x)$, meaning that σ_2 can only be downgraded by comparing it to some data at security level L.
- $m_3 \triangleq ((\sigma_1+\sigma_2)\%8)$, meaning that we can downgrade the last three bits of the sum of σ_1 and σ_2.

With these global policies, we can automatically generate the security policy for each individual secret in the following way:

Definition 7.2.1 (Label generation).
$\Sigma(\sigma_i) \triangleq \{\lambda x\!:\!\mathsf{int}.\ m_j[x/\sigma_i] \mid m_j\in\Sigma\}$

Take the example above, we have
$\Sigma(\sigma_1)=\{\lambda x\!:\!\mathsf{int}.\ x\%2,\quad \lambda x\!:\!\mathsf{int}.\ (x+\sigma_2)\%8\}$
$\Sigma(\sigma_2)=\{\lambda y\!:\!\mathsf{int}.\ \lambda x\!:\!\mathsf{int}.\ y=x,\quad \lambda x\!:\!\mathsf{int}.\ (\sigma_1+x)\%8\}$
Thus when we typecheck $\sigma_1+\sigma_2$, we can downgrade from either σ_1 or σ_2, and the results are consistent: $\lambda x\!:\!\mathsf{int}.\ x\%8$. This policy specification method not only simplifies the user's program annotation work but also make the formalization of our security guarantee more concise.

7.3 The Type System

The type system is shown in Figure 14. Compared to $\lambda^{\Downarrow}_{\mathsf{local}}$, the DLocal-L(R) rule remains unchanged. Global downgrading is supported by the DGlob-L(R) rule, which exactly shows how the labels are computed for global downgrading using information from the integrity label. All other downgrading rules are used to keep track of the integrity labels. The DTLocal and DTGlob rules are essentially the same as DLocal and DGlob, except that we compute the integrity label for the result. Integrity labels are introduced by the TSecret rule.

The SIntLabel rule patches the subtyping relation. Since our typing rules are mostly algorithmic and we do not have subsumption rules, we can make the language more convenient by changing the TCond and TOp rules to ignore integrity labels in their premises. We omitted them in this paper because they do not affect the expressiveness of the language.

$$\vdash \mathsf{int}\langle m\rangle \le \mathsf{int} \qquad \textsc{SIntLabel}$$

$$\Gamma\vdash \sigma_i : \mathsf{int}\langle\sigma_i\rangle_{\Sigma(\sigma_i)} \qquad \textsc{TSecret}$$

$$\frac{\Gamma\vdash e_1:\mathsf{int}_{l_1} \qquad \Gamma\vdash e_2:\mathsf{int}_\mathsf{L} \qquad a\triangleq\lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ x\oplus y \qquad l_1\overset{a}{\rightsquigarrow} l_3}{\Gamma\vdash e_1\oplus e_2:\mathsf{int}_{l_3}} \qquad \textsc{DLocal-L(R)}$$

$$\frac{\Gamma\vdash e_1:\mathsf{int}\langle m_1\rangle_{l_1} \qquad \Gamma\vdash e_2:\mathsf{int}_\mathsf{L} \qquad a\triangleq\lambda x\!:\!\mathsf{int}.\ \lambda y\!:\!\mathsf{int}.\ x\oplus y \qquad l_1\overset{a}{\rightsquigarrow} l_3}{\Gamma\vdash e_1\oplus e_2:\mathsf{int}\langle a\odot m_1\rangle_{l_3}} \qquad \textsc{DTLocal-L(R)}$$

$$\frac{\Gamma\vdash e_1:\mathsf{int}_{l_1} \qquad \Gamma\vdash e_2:\mathsf{int}\langle m_2\rangle_{l_2} \qquad a\triangleq\lambda x\!:\!\mathsf{int}.((\lambda y\!:\!\mathsf{int}.x\oplus y)\odot m_2) \qquad l_1\overset{a}{\rightsquigarrow} l_3}{\Gamma\vdash e_1\oplus e_2:\mathsf{int}_{l_3}} \qquad \textsc{DGlob-L(R)}$$

$$\frac{\Gamma\vdash e_1:\mathsf{int}\langle m_1\rangle_{l_1} \qquad \Gamma\vdash e_2:\mathsf{int}\langle m_2\rangle_{l_2} \qquad a\triangleq\lambda x\!:\!\mathsf{int}.\ ((\lambda y\!:\!\mathsf{int}.\ x\oplus y)\odot m_2) \qquad l_1\overset{a}{\rightsquigarrow} l_3}{\Gamma\vdash e_1\oplus e_2:\mathsf{int}\langle a\odot m_1\rangle_{l_3}} \qquad \textsc{DTGlob-L(R)}$$

Figure 14: $\lambda^{\Downarrow}_{\mathsf{global}}$ Typing Rules

7.4 The Security Goal

The security guarantee of $\lambda^{\Downarrow}_{\mathsf{global}}$ is similar to $\lambda^{\Downarrow}_{\mathsf{local}}$. The major difference is that we changed our way of policy specification. In $\lambda^{\Downarrow}_{\mathsf{global}}$, the policies are globally specified by the user: Σ is just a set of policy terms. During typechecking, Σ is split into local policies for each secret variable. Therefore, we would like to express our security goal in terms of the global policy Σ.

Theorem 7.4.1 (Relaxed Noninterference).
If $\vdash e : \mathsf{int}_{\mathsf{L}}$, *then* $\mathcal{E}(e) \equiv f m_1 \ldots m_k$ *where* $\mathsf{clean}(f)$ *and* $\forall j. m_j \in \Sigma$.

Corollary 7.4.1 (Pure Noninterference).
If $\vdash e : \mathsf{int}_{\mathsf{L}}$ *and* $\Sigma = \{\mathsf{H}\}$ *then* $\mathcal{E}(e) \equiv f$ *where* $\mathsf{clean}(f)$.

These security guarantees are similar to the ones in $\lambda^{\Downarrow}_{\mathsf{local}}$. They look even more intuitive: a safe program can only leak secrets in permitted ways, and these permissions are directly characterized by the global downgrading policy.

The proof of Theorem 7.4.1 is similar to the proof of Theorem 5.3.1. The only major difference is the reasoning about integrity labels. Lemma 7.4.1 shows the exact meaning of these integrity labels. With the help this lemma, we can go through the cases for additional downgrading rules.

Lemma 7.4.1 (Integrity guarantee).
If $\Gamma \vdash v : \mathsf{int}\langle m \rangle_l$, *then* $\mathcal{E}(\Gamma) \vdash \mathcal{E}(v) \equiv \mathsf{branch}(F_C, F)$
where $F_C = \{\mathcal{E}(v_{01}), \ldots, \mathcal{E}(v_{0i})\}$,

$$F = \{m\ \mathcal{E}(v_{11}) \ldots \mathcal{E}(v_{1k}),$$
$$\ldots\ldots\ldots$$
$$m\ \mathcal{E}(v_{j1}) \ldots \mathcal{E}(v_{jk})\}$$

and for each v_{xy}, $\Gamma \vdash v_{xy} : \mathsf{int}_{\mathsf{L}}$ *for a smaller derivation.*

Typechecking for $\lambda^{\Downarrow}_{\mathsf{global}}$ is not fundamentally harder. Handling integrity labels is algorithmic and requires no searching. The only subtle point is that the label ordering is changed in $\mathbb{L}_{\mathsf{global}}$, so we must be careful that L is not used as the lowest label in comparing labels and computing joins.

8. EVALUATION AND FUTURE WORK

Strengths and Limitations

We have presented an end-to-end style framework for downgrading policies. On one end, it provides a *policy specification* language expressive enough to represent a wide variety of downgrading policies useful in practice. On the other end, it formally describes a global *security goal* determined by the user's downgrading policy. To guarantee that a program satisfies the security goal, i.e. the program is safe with respect to the downgrading policies, we only need a *proof* showing that the program is equivalent to a specific form, by using program equivalence rules.

We also presented type systems as enforcement mechanisms. The soundness theorem of the type system ensures that, if a program is well-typed, then there exists a proof of the security goal for the program. Thus, we reduced the problem of proof searching to the problem of typechecking, which is a syntax-directed process. The programmer can explicitly write down the types as security proofs, or we can use type inference to search for proofs automatically.

It is necessary to point out that a type system is not the only possible enforcement mechanism for our framework. Type systems typically have limitations that prevent them from enforcing some kinds of downgrading policies. For example, consider the policy $\lambda x : \mathsf{int}.\ \lambda p : \mathsf{int}.\ (x + p) * p$: it cannot be enforced by our type system because typechecking is not syntax-directed. At each step, all the information is locally synthesized from adjacent nodes. For the program $(x + y) * y$ where x has the policy above, we cannot downgrade the syntax node $(x + y)$, therefore $(x + y)$ cannot have a downgradable type annotation. To reason about such policies, we need more powerful mechanisms that involve more global data-flow analysis. Nevertheless, many useful policies are not in these forms and are easily enforceable by our type system.

Understanding the Policies

The security guarantee in our framework only assures that the program respects the user's security policies, but it does not verify anything about the policies themselves. It is important to study how to evaluate the effects of these downgrading policies, especially when the program is not trusted. Both informal and formal reasoning can be used. For example, given the policy $\{\sigma_1 \% 2\}$, it is apparently true that only the parity of σ_1 can be leaked to public. Given the policy for the password: $\{\lambda y : \mathsf{int}.\ \sigma_{pwd} = y\}$, we can use the same reasoning technique in *relative secrecy* [18] and assure that any program satisfying this policy must take exponentially long expected time to crack the password. Our framework has the ability to minimize the scope of security analysis: instead of analyzing the whole program, we need to examine only the security policies for these programs, and such policies are usually several orders of magnitudes smaller than the program.

Understanding the Equivalence Relation

The equivalence rules are crucial in the definition of relaxed noninterference. Extending these rules can make the framework more expressive. For example, if we have a policy stating that $x\%4$ is safe and the equivalence relation can establish that $x\%2 \equiv (x\%4)\%2$, then $x\%2$ is also safe with respect to our policy. However, the equivalence relation must provide a useful notion of security guarantee. Take the password example again: if we use the usual definition of observational equivalence to define relaxed noninterference, it would make the following two terms equivalent:

$$\sigma_{pwd} \equiv (\mathsf{fix}\ r(x) = \mathsf{if}\ (\sigma_{pwd}{=}x)\ \mathsf{then}\ x\ \mathsf{else}\ r(x+1))\ 0$$

The consequence would be that any single occurence of the variable σ_{pwd} can be considered as a public value of type $\mathsf{int}_{\mathsf{L}}$ because it satisfies the definition of relaxed noninterference. This is apparently not a good security guarantee. Therefore, it is interesting to explore what equivalance relations are good for our purposes and how to formalize such criteria.

Practical Application

Our framework can be practically adapted into existing security-typed languages such as Jif. In our policy language, some run-time library calls and API interfaces can be modeled as operators and constants, such as encryption, primality testing and hash functions. The program annotation work mainly involves marking secret and public variables; the downgrading policies can be globally specified outside the program. In ideal cases, most type annotations can be automatically inferred during typechecking and the programmers do not need to write the downgrading policies for each piece of data in the program. To achieve this goal, more work needs to be done on type inference algorithms in our framework.

Integrating with DLM

The *decentralized label model* (DLM) expresses policies like “who can downgrade the data” and it is orthogonal to our work. Since our security policies are also formalized as a lattice of security levels, it is tempting to integrate our framework with the decentralized label model so that we can express policies like “who can downgrade the data in which ways” and achieve a better integration of access control and information flow. There has been work on combining security policies with owner information [2] in the style of DLM. This is a promising research direction we are planning to pursue in the future.

Proof Carrying Code and Information Flow

Our framework also facilitates the use of proof-carrying code for information-flow security. The downgrading policies can be specified as interfaces for untrusted software modules. The untrusted code must come with a proof showing that it respects our interfaces — in our framework, such a proof even needs not to be a typing derivation; it is sufficient to give a proof using program equivalence rules because our security goal is expressed in this way. The trusted computing base is very small: we need not trust the soundness of any type system; the correctness of our equivalence rules is almost indubitable; the proof checker is easy to implement correctly. Even without downgrading, our framework can still be very valuable in this aspect. Since we are not restricted to the use of type systems, the programmer could use more expensive proof searching techniques so that more expressive downgrading policies can be enforced.

9. CONCLUSION

In this paper, we studied the challenges of downgrading in language-based information-flow security and presented a generalized framework of downgrading policies. Such policies are treated as security levels for information flow control, specified in a simple, expressive, tractable and extensible policy language, and enforced by a type system. The security guarantee is then formalized as a concise and extensional property called *relaxed noninterference* using program equivalences, which generalizes traditional noninterference properties and accurately describes the effects of downgrading. Alternative enforcement mechanisms can also be used. Our framework now enables untrusted code to safely declassify secrets and we can guarantee that information is only leaked in permitted ways.

Acknowledgements

We would like to thank Stephen Chong, Stephen Tse, Geoffrey Washburn and the POPL reviewers for their valuable feedbacks and extensive proofreading of the original draft.

References

[1] 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.

[2] Hubie Chen and Stephen Chong. Owned policies for information security. In *Proc. of the IEEE Computer Security Foundations Workshop*, 2004.

[3] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 186–197, January 2004.

[4] 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.

[5] James W. Gray, III. Towards a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 21–34. IEEE Computer Society Press, 1991.

[6] Gavin Lowe. Quantifying information flow. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 18–31. IEEE Computer Society Press, 2002.

[7] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proceedings of The Second Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*. Springer, 2004.

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

[9] 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.

[10] 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.

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

[12] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 1–17. IEEE Computer Society Press, 2002.

[13] 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.

[14] 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.

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

[16] Andrei Sabelfeld and Andrew Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security (ISSS'03)*, 2004.

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

[18] 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.

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

[20] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Canada, June 2001. IEEE Computer Society Press.