Linear Declassification

Yûta Kaneko and Naoki Kobayashi

Graduate School of Information Sciences, Tohoku University
{kaneko,koba}@kb.ecei.tohoku.ac.jp

Abstract. We propose a new notion of declassification policy called *linear declassification*. Linear declassification controls not only which functions may be applied to declassify high-security values, but also *how often* the declassification functions may be applied. We present a linear type system which guarantees that well-typed programs never violate linear declassification policies. To state a formal security property guaranteed by the linear declassification, we also introduce *linear relaxed non-interference* as an extension of Li and Zdancewic's relaxed non-interference. An application of the linear relaxed non-interference to quantitative information flow analysis is also discussed.

1 Introduction

There have been extensive studies on policies and verification methods for information flow security [4,16,10,7,11,13]. The standard policy for secure information flow is the *non-interference* property, which means that low-security outputs cannot be affected by high-security inputs. A little more formally, a program e is secure if for any high inputs h_1 and h_2 and low input l, $e(h_1, l)$ and $e(h_2, l)$ are equivalent for low-level observers. The standard non-interference property is, however, too restricted in practice, since it does not allow any leakage of secret information. For example, a login program does leak information about the result of comparison of a string and a password.

To allow intentional release of secret information, a variety of notions of declassification have been proposed [7,12,13]. Sabelfeld and Myers [12] proposed delimited information release, where e is secure if, roughly speaking, whenever $d(h_1) = d(h_2)$ for the declassification function d, $e(h_1, l)$ and $e(h_2, l)$ are equivalent for low-level observers. As a similar criterion, Li and Zdancewic [7] proposed a notion of relaxed non-interference (relaxed NI, in short), where e is secure (i.e., satisfies relaxed NI) if $e(h, l)$ can be factorized into $e'(dh)$, where d is a declassification function and e' does not contain h. Both the frameworks guarantee that a program leaks only partial information $d(h)$ about the high-security value h. For example, if d is the function $\lambda x.x \bmod 2$, then only the parity information can be leaked.

The above criteria alone, however, do not always guarantee desirable secrecy properties. For example, consider a declassification function $d \stackrel{\triangle}{=} \lambda x.\lambda s.(s = x)$, which takes a high-security value x, and returns a *function* that takes a string and returns whether s and x are equal. Declassifications through such a function

S. Drossopoulou (Ed.): ESOP 2008, LNCS 4960, pp. 224–238, 2008.

often occur in practice, for instance, in a login program, which compares a user's password with an input string. Note that $d(h) \equiv \lambda s.(s = h)$ and h contain the same quantity of information; In fact, even if e is h itself (so that it clearly leaks the entire information), it can be factorized into:

$$(\lambda g.\textbf{let } test(s) = \textbf{if } g(s) \textbf{ then } s \textbf{ else } test(s+1) \textbf{ in } test(0))\,(d(h)).$$

Thus, the relaxed NI guarantees nothing about the quantity of information declassified through the function d. (In the case of delimited information release [12], the problem can be avoided by choosing $\lambda x.(l = x)$ as d, instead of $\lambda x.\lambda s.(s = x)$; see more detailed discussion in Section 5.)

To overcome the problem mentioned above, we propose a new notion of declassification called *linear declassification*, which controls *how often* declassification functions can be applied to each high-security value, and how often a value (which may be a function) obtained by declassification may be used. We define a linear type system that ensures that any well-typed program satisfies a given linear declassification policy.

To formalize the security property guaranteed by the linear declassification, we also extend Li and Zdancewic's relaxed non-interference [7] to *linear relaxed non-interference*, which says that e is secure if e can be factorized into $e'(\lambda^u x.(dh))$, where e' does not contain h *and* e' *can call the function* $\lambda x.(dh)$ *at most* u *times to declassify the value of* h.

The linear relaxed non-interference is useful for quantitative information flow analysis [8,3,2]. For example, if a program e containing an n-bit password satisfies the linear relaxed non-interference under the policy that $\lambda x.\lambda s.(s = x)$ is used at most once, we know that one has to run e $O(2^n)$ times in average to get complete information about the password. On the other hand, if the declassification function is replaced by $\lambda x.\lambda s.(s > x)$, the password may be leaked by only n runs of the program. In the paper, we show (through an example) that the linear relaxed non-interference enables us to estimate the quantity of information leakage (per program run) by looking at only the security policy, not the program.

The rest of this paper is structured as follows. Section 2 introduces the language of programs and linear declassification policies. Section 3 introduces a linear type system which guarantees that a program adheres to linear declassification policies. Section 4 defines linear relaxed non-interference as an extension of Li and Zdancewic's relaxed non-interference. Section 4 also discusses an application of the linear relaxed non-interference to quantitative analysis of information flow. Section 5 discusses related work and Section 6 concludes.

2 Language

This section introduces the syntax and semantics of programs and declassification policies.

2.1 Syntax

Definition 1 (expressions). The set of *expressions*, ranged over by e, is defined by:

$$\begin{aligned} e \text{ (expressions)} ::= {} & x \mid n \mid \sigma \mid d\langle\!\langle e \rangle\!\rangle \mid e_1 \oplus e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\ & \mid \lambda^u x.e \mid \mathbf{fix}\ x(y) = e \mid e_1 e_2 \mid \langle e_1, \ldots, e_n \rangle \mid \#_i(e) \\ u \text{ (uses)} ::= {} & 0 \mid 1 \mid \omega \\ \oplus \text{ (operators)} ::= {} & + \mid - \mid = \mid \cdots \end{aligned}$$

Here, the meta-variables x and n range over the sets of variables and integers respectively. The meta-variable σ ranges over the set of special variables holding high-security integers, to which security policies (given below) are associated. For the sake of simplicity, we consider only integers as primitive values, and assume that $e_1 = e_2$ returns 1 if the values of e_1 and e_2 are the same, and returns 0 otherwise. **if** e_1 **then** e_2 **else** e_3 returns the value of e_3 if the value of e_1 is 0, and returns the value of e_2 otherwise. The expression $\lambda^u x.e$ denotes a function that can be used at most u times. If u is ω, the function can be used an arbitrary number of times.[1] Note that use annotations can be automatically inferred by standard usage analysis [17,6,9], so that programmers need not specify them (except for those in policies introduced below). The expression **fix** $x(y) = e$ denotes a recursive function that can be used an arbitrary number of times. The expression $e_1 e_2$ is an ordinary function application. The expression $d\langle\!\langle e \rangle\!\rangle$ is a special form of function application, where the meta-variable d ranges over the set $\mathcal{N}_D$ of special function variables (defined in a policy introduced below). The expression $\langle e_1, \ldots, e_n \rangle$ returns a tuple consisting of the values of $e_1, \ldots, e_n$. Note that n may be 0, in which case, the tuple is empty.

We write $[e'/x]e$ for the (capture-avoiding) substitution of e' for x in e. We write $\mathbf{SVar}(e)$ for the set of security variables occurring in e.

Definition 2 (policies). The set of *policies* is defined by:

$$\begin{aligned} p \text{ (security levels)} ::= {} & \mathbf{L} \mid \mathbf{H} \mid \{d_1 \mapsto u_1, \cdots, d_n \mapsto u_n\} \\ D \text{ (declassification environment)} ::= {} & \{d_1 \mapsto \lambda^\omega x.e_1, \cdots, d_n \mapsto \lambda^\omega x.e_2\} \\ \Sigma \text{ (policy)} ::= {} & \{\sigma_1 \mapsto p_1, \cdots, \sigma_n \mapsto p_n\} \end{aligned}$$

A security level p expresses the degree of confidentiality of each value. If p is $\mathbf{L}$, the value may be leaked to low-security principals. If p is $\mathbf{H}$, no information about the value may be leaked. If p is $\{d_1 \mapsto u_1, \cdots, d_n \mapsto u_n\}$, then the value may be leaked only through declassification functions $d_1, \ldots, d_n$ and each declassification function d_i may be applied to the value at most u_i times. For example, if the security level of σ is $\{d_1 \mapsto 1, d_2 \mapsto \omega, d_3 \mapsto 0\}$, then $d_1\langle\!\langle\sigma\rangle\!\rangle + d_2\langle\!\langle\sigma\rangle\!\rangle + d_2\langle\!\langle\sigma\rangle\!\rangle$ is allowed, but neither $d_3\langle\!\langle\sigma\rangle\!\rangle$ nor $d_1\langle\!\langle\sigma\rangle\!\rangle + d_1\langle\!\langle\sigma\rangle\!\rangle$ is.

A declassification environment D defines declassification functions. A policy Σ maps σ_i to its security level. Note that the use of $D(d_i)$ is always ω. This is because how often d_i can be used is described in Σ for each security variable σ.

[1] For the sake of simplicity, we consider only $0, 1, \omega$ as uses. It is easy to extend the language and the type system given in the next section to allow $2, 3, \ldots$.

Example 1. Let $D = \{d \mapsto \lambda^{\omega}x.\lambda^{1}y.x = y\}$ and $\Sigma = \{\sigma \mapsto \{d \mapsto 1\}\}$. This policy specifies that information about σ can be leaked by at most one application of d. Since the result of the application is a linear (use-once) function $\lambda^{1}y.\sigma = y$, the policy means that σ may be compared with another integer only once.

Note that if $D(d)$ is $\lambda^{\omega}x.\lambda^{\omega}y.x = y$, then the declassification may be performed only once, but the resulting value $\lambda^{\omega}y.\sigma = y$ can be used an arbitrary number of times. Therefore, an attacker can obtain complete information about σ by applying the function to different values.

2.2 Operational Semantics

This section introduces an operational semantics to define the meaning of expressions and policies formally.

A run-time state is modeled by a pair $\langle H, e\rangle$, where H is a heap given below.[2]

Definition 3 (heap)

$$\begin{aligned} H \text{ (heap)} ::= {} & \{f_1 \mapsto \lambda^{u_1}x_1.e_1, \ldots, f_n \mapsto \lambda^{u_n}x_n.e_n, \\ & \sigma_1 \mapsto (n_1, p_1), \ldots, \sigma_m \mapsto (n_m, p_m)\} \\ f \text{ (function pointer)} ::= {} & x \mid d \end{aligned}$$

Here, f ranges over the set consisting of (ordinary) variables $(x, y, z, \ldots)$ and declassification function variables $(d_1, d_2, \ldots,)$.

A heap H keeps information about how often each function may be applied and how the value of each security variable may be declassified in the rest of the computation. For example, $H(\sigma) = (2, \{d \mapsto 1\})$ means that the value of σ is 2, and the value can be declassified only once through the declassification function d.

For a system (Σ, D, e), the initial heap is determined by Σ, D, and the actual values of the security variables. Let g be a mapping from $dom(\Sigma)$ to the set of integers. We write $H_{\Sigma,D,g}$ for the initial heap $D \cup \{\sigma_1 \mapsto (g(\sigma_1), \Sigma(\sigma_1)), \ldots, \sigma_k \mapsto (g(\sigma_k), \Sigma(\sigma_k))\}$ (where $dom(\Sigma) = \{\sigma_1, \ldots, \sigma_k\}$). We use evaluation contexts to define the operational semantics.

Definition 4 (evaluation context). The set of evaluation contexts, ranged over by E, is given by:

$$\begin{aligned} E \text{ (evaluation context)} ::= {} & [\,] \mid [\,]e \mid x[\,] \mid d\langle\!\langle [\,] \rangle\!\rangle \mid \textbf{if } [\,] \textbf{ then } e_1 \textbf{ else } e_2 \\ & \mid [\,] \oplus e \mid v \oplus [\,] \mid \langle v_1, \ldots, v_{k-1}, [\,], e_{k+1}, \ldots, e_n\rangle \mid \#_i([\,]) \\ v \text{ (values)} ::= {} & f \mid n \mid \sigma \mid \langle v_1, \ldots, v_n\rangle \end{aligned}$$

The relation $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$ is the least relation closed under the rules in Figure 1. In the figure, $F\{x \mapsto v\}$ is the mapping F' such that $F'(x) = v$, and $F'(y) = F(y)$ for any $y \in dom(F) \setminus \{x\}$. $val(H, v)$ is defined to be n if $v = n$, or $v = \sigma$ and $H(\sigma) = (n, p)$.

The key rules are E-App and E-Decl. In E-App, the use of the function y is decreased by one. Here, the subtraction $u - 1$ is defined by: $1 - 1 = 0$ and

[2] Note that unlike the usual heap-based semantics, tuples are *not* stored in a heap.

$$\frac{y \text{ fresh}}{\langle H, E[\lambda^u x.e]\rangle \longrightarrow \langle H\{y \mapsto \lambda^u x.e\}, E[y]\rangle} \quad \text{(E-Fun)}$$

$$\frac{H(d) = \lambda^\omega x.e}{\langle H\{\sigma \mapsto (n,p)\}, E[d\langle\!\langle\sigma\rangle\!\rangle]\rangle \longrightarrow \langle H\{\sigma \mapsto (n, p-d)\}, E[[n/x]e]\rangle} \quad \text{(E-Decl)}$$

$$\frac{H(d) = \lambda^\omega x.e}{\langle H, E[d\langle\!\langle n\rangle\!\rangle]\rangle \longrightarrow \langle H, E[[n/x]e]\rangle} \quad \text{(E-Decl2)}$$

$$\langle H\{y \mapsto \lambda^u x.e\}, E[yv]\rangle \longrightarrow \langle H\{y \mapsto \lambda^{u-1} x.e\}, E[[v/x]e]\rangle \quad \text{(E-App)}$$

$$\frac{val(H,v) \neq 0}{\langle H, E[\textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2]\rangle \longrightarrow \langle H, E[e_1]\rangle} \quad \text{(E-IfT)}$$

$$\frac{val(H,v) = 0}{\langle H, E[\textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2]\rangle \longrightarrow \langle H, E[e_2]\rangle} \quad \text{(E-IfF)}$$

$$\langle H, E[v_1 \oplus v_2]\rangle \longrightarrow \langle H, E[val(H,v_1) \underline{\oplus} val(H,v_2)]\rangle \quad \text{(E-Op)}$$

$$\frac{z \text{ fresh}}{\langle H, E[\textbf{fix } x(y) = e]\rangle \longrightarrow \langle H \cup \{z \mapsto \lambda^\omega y.[z/x]e\}, E[z]\rangle} \quad \text{(E-Fix)}$$

$$\langle H, E[\#_i\langle v_1, \ldots, v_n\rangle]\rangle \longrightarrow \langle H, E[v_i]\rangle \quad \text{(E-Proj)}$$

Fig. 1. Evaluation rules

$\omega - 1 = \omega$. Note that $0 - 1$ is undefined, so that if $H(y) = \lambda^0 x.e$, the function y can no longer be used (in other words, the evaluation of $E[yv]$ get stuck).

In E-Decl, the security level p for σ changes after the reduction. Here, $p - d$ is defined by:

$$\{d_1 \mapsto u_1, \ldots, d_n \mapsto u_n\} - d_i = \{d_1 \mapsto u'_1, \ldots, d_n \mapsto u'_n\}$$
$$\text{where } u'_j = \begin{cases} u_j - 1 \text{ if } j = i \\ u_j \quad \text{otherwise} \end{cases}$$
$$\mathbf{L} - d_i = \mathbf{L}$$

For example, if the security level p of σ is $\{d \mapsto 1\}$, then after the declassification, the security level becomes $p - d = \{d \mapsto 0\}$, which means that the value of σ can no longer be declassified. Note that $\mathbf{H} - d_i$ is undefined, so that an integer of security level $\mathbf{H}$ can never be declassified. Rule E-Decl2 is for the case when a declassification function d is applied to an ordinary integer.

In rule E-Op, $\underline{\oplus}$ is the binary operation on integers denoted by the operator symbol $\oplus$. The remaining rules are standard.

Example 2. Recall the security policy in Example 1: $D = \{d \mapsto \lambda^\omega x.\lambda^1 y.(x = y)\}$ and $\Sigma = \{\sigma \mapsto \{d \mapsto 1\}\}$.
$\langle H_{\Sigma,D,\{\sigma\mapsto 3\}}, d\langle\!\langle\sigma\rangle\!\rangle 2\rangle$ is reduced as follows.

$$\begin{aligned} &\langle D \cup \{\sigma \mapsto (3, \{d \mapsto 1\})\}, d\langle\!\langle\sigma\rangle\!\rangle 2\rangle \\ \longrightarrow\ &\langle D \cup \{\sigma \mapsto (3, \{d \mapsto 0\})\}, (\lambda^1 y.(3 = y))2\rangle \\ \longrightarrow\ &\langle D \cup \{\sigma \mapsto (3, \{d \mapsto 0\}), z \mapsto \lambda^1 y.(3 = y)\}, z(2)\rangle \end{aligned}$$

$$\longrightarrow \langle D \cup \{\sigma \mapsto (3, \{d \mapsto 0\}), z \mapsto \lambda^0 y.(3 = y)\}, 3 = 2\rangle$$
$$\longrightarrow \langle D \cup \{\sigma \mapsto (3, \{d \mapsto 0\}), z \mapsto \lambda^0 y.(3 = y)\}, 0\rangle$$

On the other hand, both $\langle d\langle\!\langle\sigma\rangle\!\rangle, d\langle\!\langle\sigma\rangle\!\rangle\rangle$ and $(\lambda^\omega f.\langle f(1), f(2)\rangle)(d\langle\!\langle\sigma\rangle\!\rangle)$ get stuck. □

3 Type System

This section introduces a linear type system, which ensures that if $\langle \Sigma, D, e\rangle$ is well-typed, then e satisfies the security policy specified by Σ and D.

3.1 Types

Definition 5 (types). The set of types, ranged over by τ, is defined by:

$$\tau \text{ (types)} ::= int_p \mid \tau_1 \xrightarrow{\varphi}_u \tau_2 \mid \langle \tau_1, \ldots, \tau_n\rangle$$
$$\varphi \text{ (effects)} ::= \mathtt{t} \mid \mathtt{nt}$$

The integer type int_p describes integers whose security level is p. For example, $int_{\{d\mapsto 1\}}$ is the type of integers that can be declassified through the function d at most once. The function type $\tau_1 \xrightarrow{\varphi}_u \tau_2$ describes functions that can be used at most u times and that take a value of type τ_1 as an argument and return a value of type τ_2. The effect φ describes whether the function is terminating (when $\varphi = \mathtt{t}$) or it may not be terminating (when $\varphi = \mathtt{nt}$). The effect will be used for preventing leakage of information from the termination behavior of a program. The type $\langle \tau_1, \ldots, \tau_n\rangle$ describes tuples consisting of values of types $\tau_1, \ldots, \tau_n$.

The *sub-effect relation* $\leq$ on effects is the partial order defined by $\mathtt{t} \leq \mathtt{nt}$. The *sub-level relation* $\sqsubseteq$ on security levels and the subtyping relation $\tau_1 \leq \tau_2$ are the least relations closed under the rules in Figure 2. For example, $int_{\{d\mapsto 1\}} \xrightarrow{\mathtt{t}}_\omega int_{\{d\mapsto\omega\}}$ is a subtype of $int_{\{d\mapsto\omega\}} \xrightarrow{\mathtt{nt}}_1 int_{\{d\mapsto 1\}}$. We write $\varphi_1 \vee \varphi_2$ for the least upper bound of φ_1 and φ_2 (with respect to $\leq$), and $p_1 \sqcup p_2$ for the least upper bound of p_1 and p_2 with respect to $\sqsubseteq$.

$$\frac{}{\mathbf{L} \sqsubseteq p \sqsubseteq \mathbf{H}} \qquad \frac{u'_i \leq u_i \text{ for each } i \in \{1, \ldots, m\}}{\{d_1 \mapsto u_1, \ldots, d_m \mapsto u_m, \ldots\} \sqsubseteq \{d_1 \mapsto u'_1, \ldots, d_m \mapsto u'_m\}}$$

$$\frac{p_1 \sqsubseteq p_2}{int_{p_1} \leq int_{p_2}} \qquad \frac{\tau'_1 \leq \tau_1 \qquad \tau_2 \leq \tau'_2 \qquad u' \leq u \qquad \varphi \leq \varphi'}{\tau_1 \xrightarrow{\varphi}_u \tau_2 \leq \tau'_1 \xrightarrow{\varphi'}_{u'} \tau'_2}$$

$$\frac{\tau_i \leq \tau'_i \text{ for each } i \in \{1, \ldots, n\}}{\langle \tau_1, \ldots, \tau_n\rangle \leq \langle \tau'_1, \ldots, \tau'_n\rangle}$$

Fig. 2. Subtyping rules

3.2 Typing

A type environment is a mapping from a finite set consisting of extended variables (ordinary variables, security variables, and declassification function names) to types. We have two forms of type judgment: $\vdash \langle \Sigma, D, e\rangle$ for the whole system (consisting of a policy, a declassification environment, and an expression), and $\Gamma \vdash e : \tau \,\&\, \varphi$ for expressions. The judgment $\vdash \langle \Sigma, D, e\rangle$ means that e satisfies the security policy specified by Σ and D. $\Gamma \vdash e : \tau \,\&\, \varphi$ means that e evaluates to a value of type τ under an environment described by Γ. If $\varphi = \mathtt{t}$, then evaluation of e must terminate. If $\varphi = \mathtt{nt}$, then e may or may not terminate. For example, $\sigma : int_{\{d\mapsto 1\}}, f : int_{\{d\mapsto 1\}} \xrightarrow{\mathtt{t}}_{\omega} int_{\{d\mapsto 1\}} \vdash f\sigma : int_{\{d\mapsto 1\}} \,\&\, \mathtt{t}$ is a valid judgment, but neither $\sigma : int_{\{d\mapsto 1\}}, f : int_{\{d\mapsto \omega\}} \xrightarrow{\mathtt{t}}_{\omega} int_{\{d\mapsto 1\}} \vdash f\sigma : int_{\{d\mapsto 1\}} \,\&\, \mathtt{t}$ nor $\sigma : int_{\{d\mapsto 1\}}, f : int_{\{d\mapsto 1\}} \xrightarrow{\mathtt{nt}}_{\omega} int_{\{d\mapsto 1\}} \vdash f\sigma : int_{\{d\mapsto 1\}} \,\&\, \mathtt{t}$ is. (In the former, the security level of σ does not match that of the argument required by f. In the latter, the type of f says that f may not terminate, but the conclusion says that $f\sigma$ terminates.)

Figure 3 shows the typing rules. Two auxiliary judgments $\vdash \Sigma : \Gamma$ and $\vdash D : \Gamma$ are used for defining $\vdash \langle \Sigma, D, e\rangle$. The definitions of the operations used in the typing rules are summarized in Figure 4.

We explain some key rules below.

- T-Op: Suppose e_1 has type $int_{\{d\mapsto 1\}}$. Then, the value of e_1 can be declassified through the function d, but that does not necessarily imply that $e_1 \oplus e_2$ can be declassified through the function d. Therefore, we raise the security level of $e_1 \oplus e_2$ to $\mathbf{H}$ unless both of the security levels of e_1 and e_2 are $\mathbf{L}$.
- T-If: Since information about the value of e_0 indirectly flows to the value of the if-expression, the security level of the if-expression should be greater than or equal to the *ceil* of security level of e_0. For the sake of simplicity, we require that the values of if-expressions must be integers.
- T-Fun: The premise means that free variables are used according to Γ *each time* the function is applied. Since the function may be applied u times, the usage of free variables is expressed by $u \cdot \Gamma$ in total.
- T-Dcl: The premise ensures that e must have type $int_{d\mapsto 1}$, so that e can indeed be declassified through d.

Example 3. Let $\tau_d = int_{\mathbf{L}} \xrightarrow{\mathtt{t}}_{\omega} int_{\mathbf{L}} \xrightarrow{\mathtt{t}}_{1} int_{\mathbf{L}}$. $d\langle\!\langle\sigma\rangle\!\rangle 2$ is typed as follows.

$$\dfrac{\dfrac{\sigma : int_{\{d\mapsto 1\}} \vdash \sigma : int_{\{d\mapsto 1\}} \,\&\, \mathtt{t}}{d : \tau_d, \sigma : int_{\{d\mapsto 1\}} \vdash d\langle\!\langle\sigma\rangle\!\rangle : int_{\mathbf{L}} \xrightarrow{\mathtt{t}}_{1} int_{\mathbf{L}} \,\&\, \mathtt{t}} \quad \emptyset \vdash 2 : int_{\mathbf{L}} \,\&\, \mathtt{t}}{d : \tau_d, \sigma : int_{\{d\mapsto 1\}} \vdash d\langle\!\langle\sigma\rangle\!\rangle 2 : int_{\mathbf{L}} \,\&\, \mathtt{t}}$$

Example 4. Let e be $\mathbf{fix}\, f(x) = \mathbf{if}\ d\langle\!\langle\sigma\rangle\!\rangle x\ \mathbf{then}\ x\ \mathbf{else}\ f(x+1)$. Let $\Sigma_1 = \{\sigma \mapsto \{d \mapsto \omega\}\}$, $\Sigma_2 = \{\sigma \mapsto \{d \mapsto 1\}\}$, and $D = \{d \mapsto \lambda^{\omega} x.\lambda^{1} y.(x = y)\}$. Then, $\vdash \langle \Sigma_1, D, e(0)\rangle : int_{\mathbf{L}}$ holds but $\vdash \langle \Sigma_2, D, e(0)\rangle : int_{\mathbf{L}}$ does not.

$\Gamma \vdash e : \tau$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \,\&\, \mathtt{t}} \quad \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash n : int_{\mathbf{L}} \,\&\, \mathtt{t}} \quad \text{(T-Const)}$$

$$\frac{}{\Gamma, \sigma : int_p \vdash \sigma : int_p \,\&\, \mathtt{t}} \quad \text{(T-SVal)}$$

$$\frac{\Gamma_1 \vdash e_1 : int_{p_1} \,\&\, \varphi \qquad \Gamma_2 \vdash e_2 : int_{p_2} \,\&\, \varphi}{\Gamma_1 + \Gamma_2 \vdash e_1 \oplus e_2 : int_{\lceil p_1 \rceil \sqcup \lceil p_2 \rceil} \,\&\, \varphi} \quad \text{(T-Op)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \varphi}{u \cdot \Gamma \vdash \lambda^u x.e : \tau_1 \xrightarrow{\varphi}_u \tau_2 \,\&\, \mathtt{t}} \quad \text{(T-Fun)}$$

$$\frac{\Gamma, x : \tau_1 \xrightarrow{\mathtt{nt}}_\omega \tau_2, y : \tau_1 \vdash e : \tau_2 \,\&\, \varphi}{\omega \cdot \Gamma \vdash \mathbf{fix}\ x(y) = e : \tau_1 \xrightarrow{\varphi}_\omega \tau_2 \,\&\, \mathtt{t}} \quad \text{(T-Fix)}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \xrightarrow{\varphi_0}_1 \tau_2 \,\&\, \varphi_1 \qquad \Gamma_2 \vdash e_2 : \tau_1 \,\&\, \varphi_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau_2 \,\&\, \varphi_0 \vee \varphi_1 \vee \varphi_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash e : int_{\{d \mapsto 1\}} \,\&\, \varphi_1}{(d : int_{\mathbf{L}} \xrightarrow{\varphi_0}_\omega \tau) + \Gamma \vdash d\langle\!\langle e \rangle\!\rangle : \tau \,\&\, \varphi_0 \vee \varphi_1} \quad \text{(T-Dcl)}$$

$$\frac{\Gamma \vdash e : \tau' \,\&\, \varphi' \qquad \tau' \leq \tau \qquad \varphi' \leq \varphi}{\Gamma \vdash e : \tau \,\&\, \varphi} \quad \text{(T-Sub)}$$

$$\frac{\Gamma_1 \vdash e_0 : int_{p_0} \,\&\, \varphi_0 \qquad \Gamma_2 \vdash e_1 : int_{p_1} \,\&\, \varphi_1 \qquad \Gamma_2 \vdash e_2 : int_{p_2} \,\&\, \varphi_2 \qquad \varphi_1 = \varphi_2 = \mathtt{t} \text{ if } \lceil p_0 \rceil = \mathbf{H}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : int_{\lceil p \rceil \sqcup p_1 \sqcup p_2} \,\&\, \varphi_0 \vee \varphi_1 \vee \varphi_2} \quad \text{(T-If)}$$

$$\frac{\Gamma_i \vdash e_i : \tau_i \,\&\, \varphi_i \text{ (for each } i \in \{1, \ldots, n\})}{\Gamma_1 + \cdots + \Gamma_n \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle \,\&\, \varphi_1 \vee \cdots \vee \varphi_n} \quad \text{(T-Tuple)}$$

$\vdash \Sigma : \Gamma$

$$\vdash \{\sigma_1 \mapsto p_1, \ldots, \sigma_n \mapsto p_n\} : (\sigma_1 : int_{p_1}, \ldots, \sigma_n : int_{p_n}) \quad \text{(T-Policy)}$$

$\vdash D : \Gamma$

$$\frac{\emptyset \vdash \lambda^\omega x.e_i : \tau_i \,\&\, \varphi_i \text{ for each } i \in \{1, \ldots, n\}}{\vdash \{d_1 \mapsto \lambda^\omega x.e_1, \cdots, d_n \mapsto \lambda^\omega x.e_n\} : (d_1 : \tau_1, \ldots, d_n : \tau_n)} \quad \text{(T-DEnv)}$$

$\vdash \langle \Sigma, D, e \rangle$

$$\frac{\vdash \Sigma : \Gamma_1 \qquad \vdash D : \Gamma_2 \qquad \Gamma_1, \Gamma_2 \vdash e : \tau \,\&\, \varphi \qquad \text{all the security levels in } \Gamma_2 \text{ are } \mathbf{L}}{\vdash \langle \Sigma, D, e \rangle : \tau} \quad \text{(T-Sys)}$$

Fig. 3. Typing rules

3.3 (Partial) Type Soundness

The following theorem means that evaluation of a well-typed program never gets stuck. A proof is given in the extended version of this paper [5].

$$u_1 + u_2 = \begin{cases} 0 \text{ if } u_1 = u_2 = 0 \\ 1 \text{ if } (u_1, u_2) \in \{(0,1),(1,0)\} \\ \omega \text{ otherwise} \end{cases}$$

$int_{\mathbf{L}} + int_{\mathbf{L}} = int_{\mathbf{L}} \qquad int_{\mathbf{H}} + int_{\mathbf{H}} = int_{\mathbf{H}}$

$int_{\{d_1 \mapsto u_1, \ldots, d_n \mapsto u_n\}} + int_{\{d_1 \mapsto u'_1, \ldots, d_n \mapsto u'_n\}} = int_{\{d_1 \mapsto (u_1 + u'_1), \ldots, d_n \mapsto (u_n + u'_n)\}}$

$(\tau_1 \xrightarrow{\varphi}_u \tau_2) + (\tau_1 \xrightarrow{\varphi}_{u'} \tau_2) = \tau_1 \xrightarrow{\varphi}_{(u+u')} \tau_2$

$\langle \tau_1, \ldots, \tau_n \rangle + \langle \tau'_1, \ldots, \tau'_n \rangle = \langle \tau_1 + \tau'_1, \ldots, \tau_n + \tau'_n \rangle$

$$(\Gamma_1 + \Gamma_2)\ (x) = \begin{cases} \Gamma_1\ (x) & \text{if } x \in dom(\Gamma_1) \setminus dom(\Gamma_2) \\ \Gamma_2\ (x) & \text{if } x \in dom(\Gamma_2) \setminus dom(\Gamma_1) \\ \Gamma_1\ (x) + \Gamma_2\ (x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \end{cases}$$

$$u_1 \cdot u_2 = \begin{cases} 0 \text{ if } u_1 = 0 \text{ or } u_2 = 0 \\ 1 \text{ if } u_1 = u_2 = 1 \\ \omega \text{ otherwise} \end{cases}$$

$u \cdot int_L = int_L \qquad u \cdot int_H = int_H$

$u \cdot int_{\{d_1 \mapsto u_1, \ldots, d_n \mapsto u_n\}} = int_{\{d_1 \mapsto u \cdot u_1, \ldots, d_n \mapsto u \cdot u_n\}}$

$u \cdot (\tau_1 \to_{u'} \tau_2) = \tau_1 \to_{u \cdot u'} \tau_2 \qquad u \cdot \langle \tau_1, \ldots, \tau_n \rangle = \langle u \cdot \tau_1, \ldots, u \cdot \tau_n \rangle$

$(u \cdot \Gamma)\ (x) = u \cdot \Gamma(x)$

$$\lceil p \rceil = \begin{cases} \mathbf{L} \text{ if } p = \mathbf{L} \\ \mathbf{H} \text{ otherwise} \end{cases}$$

Fig. 4. Operations on policies, types, and type environments

Theorem 1. *Suppose that* $dom(\Sigma) = \{\sigma_1, \ldots, \sigma_k\}$.
If $\vdash \langle \Sigma, D, e \rangle$ *and* $\langle H_{\Sigma,D,\{\sigma_1 \mapsto n_1, \ldots, \sigma_k \mapsto n_k\}}, e \rangle \longrightarrow^* \langle H, e' \rangle \not\longrightarrow$, *then* e' *is a value.*

Note that Theorem 1 alone does not necessarily guarantee that e satisfies the security policy. In fact, the evaluation of $\langle H_{\emptyset,\emptyset,\{\sigma \mapsto 2\}}, \sigma + 1 \rangle$ does not get stuck (yields the value 3), but it does leak information about σ. The security property satisfied by well-typed programs is formalized in the next section.

4 Linear Relaxed Non-interference

In this section, we define *linear relaxed non-interference* (linear relaxed NI, in short) as a new criterion of information flow security, and prove that well-typed programs of our type system satisfy that criterion. Linear relaxed NI is an extension of relaxed NI [7]. We first review relaxed NI and discuss its weakness in Section 4.1. We then define linear relaxed NI and show that our type system guarantees linear relaxed NI. Section 4.3 discusses an application of linear relaxed NI to quantitative information flow analysis.

4.1 Relaxed Non-interference

Relaxed non-interference [7] is an extension of non-interference. Suppose that $\Sigma = \{\sigma \mapsto \{d \mapsto \omega\}\}$. Informally, an expression e satisfies relaxed NI under the

policy Σ if e can be factorized (up to a certain program equivalence) into $e'(d\sigma)$, where e' does not contain σ. If d is a constant function $\lambda x.0$, then the relaxed NI degenerates into the standard non-interference.

As already discussed in Section 1, the relaxed NI does not always guarantee a desired secrecy property. For example, consider the case where $d = \lambda x.\lambda y.x = y$. Then, *any* expression containing σ can be factorized into $e'(d\sigma)$ up to the standard contextual equivalence. In fact, σ is contextually-equivalent to:[3]

$$(\lambda^{\omega} g.(\mathbf{fix}\ test(s) = \mathbf{if}\ g(s)\ \mathbf{then}\ s\ \mathbf{else}\ test(s+1))\ 0)(d\langle\!\langle\sigma\rangle\!\rangle)$$

4.2 Linear Relaxed Non-interference

We first define the notion of (typed) contextual equivalence. For the sake of simplicity, we consider only closed terms (thus, it suffices to consider only contexts of the form $e[\,]$). We write $\langle H, e\rangle \Downarrow n$ if $\langle H, e\rangle \longrightarrow^* \langle H', n\rangle$ for some n.

Definition 6 (contextual equivalence). Suppose that $\emptyset \vdash e_1 : \tau\,\&\,\varphi$ and $\emptyset \vdash e_2 : \tau\,\&\,\varphi$. e_1 and e_2 are *contextually equivalent*, written $e_1 \approx_{\tau,\varphi} e_2$, if, for any e such that $\emptyset \vdash e : \tau \xrightarrow{\mathtt{nt}}_{\omega} int_{\mathbf{L}}$, $\langle \emptyset, ee_1\rangle \Downarrow 0$ if and only if $\langle \emptyset, ee_2\rangle \Downarrow 0$.

Note that in the above definition, the initial heap is empty, so that neither security variables σ nor declassification functions are involved; thus, the contextual equivalence above should coincide with standard typed equivalence for linear λ-calculus.

We now define the linear relaxed non-interference.

Definition 7 (linear relaxed non-interference). Let $\Sigma = \{\sigma_1 \mapsto \{d_1 \mapsto u_{11}, \ldots, d_k \mapsto u_{1k}\}, \ldots, \sigma_m \mapsto \{d_1 \mapsto u_{m1}, \ldots, d_k \mapsto u_{mk}\}\}$. Suppose also that $\mathbf{SVar}(e) \subseteq \{\sigma_1, \ldots, \sigma_m\}$. $\langle \Sigma, D, e\rangle$ satisfies *linear relaxed non-interference* at τ if there exists e' such that the following equivalence holds for any integers $n_1, \ldots, n_m$:

$$[n_1/\sigma_1, \ldots, n_m/\sigma_m]D(e) \approx_{\tau,\mathtt{nt}} e'\ \begin{array}{l}\langle \lambda^{u_{11}} x.(D(d_1)n_1), \ldots, \lambda^{u_{1k}} x.(D(d_k)n_1)\rangle \\ \cdots \\ \langle \lambda^{u_{m1}} x.(D(d_1)n_m), \ldots, \lambda^{u_{mk}} x.(D(d_k)n_m)\rangle\end{array}$$

Here $D(e)$ denotes the term obtained from e by replacing each occurrence of a declassification expression $d\langle\!\langle e\rangle\!\rangle$ with $D(d)e$.

Intuitively, the above definition means that if $\langle \Sigma, D, e\rangle$ satisfies *linear relaxed non-interference*, then e can leak information about the security variables $\sigma_1, \ldots, \sigma_m$ only by calling declassification functions at most the number of times specified by Σ. Note that in the above definition, e' cannot depend on the values of the security variables $n_1, \ldots, n_m$.

[3] Actually, Li and Zdancewic [7] uses a finer equivalence than the contextual equivalence, so that the above factorization is not valid. However, if σ ranges over a finite set, then a similar factorization is possible by unfolding the recursion: consider a program $\mathbf{if}\ \sigma = 0\ \mathbf{then}\ 0\ \mathbf{else\ if}\ \sigma = 1\ \mathbf{then}\ 1\ \mathbf{else\ if}\ \sigma = 2\ \mathbf{then}\ 2\ \mathbf{else} \cdots$.

We now show that well-typed programs satisfy linear relaxed non-interference.

Theorem 2. *If* $\vdash \langle \Sigma, D, e\rangle : \tau$ *and all the security levels in* τ *are* $\mathbf{L}$*, then* $\langle \Sigma, D, e\rangle$ *satisfies the linear relaxed non-interference at* τ*.*

A proof of the above theorem is given in [5].

4.3 Application to Quantitative Information Flow Analysis

In this subsection, we discuss how linear relaxed NI can be applied to quantitative information flow analysis [8,2]. Unlike the classical information flow analysis, which obtains *binary* information of whether or not a high-security value is leaked to public, the quantitative analysis aims to estimate the *quantity* of the information leakage based. Recently, definitions and methods of the quantitative information flow analysis have been extensively studied by Malacaria et al. [8,2], based on Shannon's information theory [15]. The quantitative analysis is generally more expensive than the classical information flow analysis, and has not been fully automated. As discussed below, the linear relaxed NI enables us to estimate the quantity of information leakage per program run *by looking at only the security policy*, not the program itself. Since the security policy of a program is typically much smaller than the program itself, this reduces the cost of quantitative information flow analysis.

For the sake of simplicity, we consider below only a single high security variable σ and the declassification environment $D = \{d \mapsto \lambda^{\omega} x.\lambda^{1} y.x \oplus y\}$, with the fixed security policy $\Sigma = \{\sigma \mapsto \{d \mapsto 1\}\}$.

Suppose that $\langle \Sigma, D, e\rangle$ satisfies linear relaxed NI at $int_{\mathbf{L}}$. Let us consider the *quantity* of information that flows from σ to the value of e. By Definition 7, there exists an e' such that for any n and n_1, $\langle \{\sigma \mapsto (n,p)\} \cup D, e\rangle \Downarrow n_1$ if and only if $\langle \{\sigma \mapsto (n,p)\} \cup D, e'\langle \lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle\rangle \rangle \Downarrow n_1$, where e' does not contain σ. Moreover, since $e'(\lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle)$ is well-typed, if $\langle \{\sigma \mapsto (n,p)\} \cup D, e'\langle \lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle\rangle \rangle \longrightarrow^* \langle H, n_1\rangle$ and the value of σ is used during the reduction, then the reduction sequence must be of the following form:[4]

$$
\begin{aligned}
& \langle \{\sigma \mapsto (n, \{d \mapsto 1\})\} \cup D, e'\langle \lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle\rangle \rangle \\
\longrightarrow^* & \langle \{\sigma \mapsto (n, \{d \mapsto 1\})\} \cup H_1, E_1[\lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle] \rangle \\
\longrightarrow^* & \langle \{\sigma \mapsto (n, \{d \mapsto 1\}), z \mapsto \lambda^1 x.d\langle\!\langle \sigma \rangle\!\rangle\} \cup H_2, E_2[z\langle\,\rangle] \rangle \\
\longrightarrow^* & \langle \{\sigma \mapsto (n, \{d \mapsto 0\}), z \mapsto \lambda^0 x.d\langle\!\langle \sigma \rangle\!\rangle, w \mapsto \lambda^1 y.n \oplus y\} \cup H_3, E_3[w(m)] \rangle \\
\longrightarrow & \langle \{\sigma \mapsto (n, \{d \mapsto 0\}), z \mapsto \lambda^0 x.d\langle\!\langle \sigma \rangle\!\rangle, w \mapsto \lambda^0 y.n \oplus y\} \cup H_3, E_3[n \oplus m] \rangle \\
\longrightarrow & \langle \{\sigma \mapsto (n, \{d \mapsto 0\}), z \mapsto \lambda^0 x.d\langle\!\langle \sigma \rangle\!\rangle, w \mapsto \lambda^0 y.n \oplus y\} \cup H_3, E_3[m'] \rangle \\
\longrightarrow^* & \langle \{\sigma \mapsto (n, \{d \mapsto 0\}), z \mapsto \lambda^0 x.d\langle\!\langle \sigma \rangle\!\rangle, w \mapsto \lambda^0 y.n \oplus y\} \cup H_4, n_1 \rangle
\end{aligned}
$$

Here, since e' does not contain σ, H_i and E_i $(i = 1, 2, 3)$ are independent of the value n of σ.

[4] For the sake of simplicity, we consider only terminating programs. Non-terminating programs can be treated in a similar manner, by introducing a special value $\perp$ for representing non-termination.

Let $\mathtt{L}$ be a random variable representing e' above, $\mathtt{H}$ be a random variable representing the value n of σ, and $\mathtt{O}$ be a random variable representing the final value n_1. Then, by the reduction sequence above, $\mathtt{O}$ can be expressed as follows.

$$\mathtt{O} = f_0(f_1(\mathtt{L}), \mathtt{H} \underline{\oplus} f_2(\mathtt{L}))$$

Here, $f_1(\mathtt{L})$ corresponds to the pair (H_3, E_3) and $f_2(\mathtt{L})$ corresponds to m in the reduction step above. The function f_0 represents the computation of n_1 from the configuration $\langle\{\sigma \mapsto (n, \{d \mapsto 0\}), \ldots\} \cup H_3, E_3[m']\rangle$.

According to [8,2], the leakage of information is expressed by:[5]

$$\mathcal{I}(\mathtt{O}; \mathtt{H} \mid \mathtt{L}) = \mathcal{H}(\mathtt{O} \mid \mathtt{L}) = \mathcal{H}(\mathtt{O}, \mathtt{L}) - \mathcal{H}(\mathtt{L})$$

Here, $\mathcal{H}(\vec{X})$ is defined as $\Sigma_x P(\vec{X} = \vec{x}) \log \frac{1}{P(\vec{X}=\vec{x})}$ (and $P(\vec{X} = \vec{x})$ denotes the probability that the value of $\vec{X}$ is $\vec{x}$).

Using $\mathtt{O} = f_0(f_1(\mathtt{L}), \mathtt{H} \underline{\oplus} f_2(\mathtt{L}))$, $\mathcal{I}(\mathtt{O}; \mathtt{H} \mid \mathtt{L})$ is estimated as follows.

$$\begin{aligned}
\mathcal{I}(\mathtt{O}; \mathtt{H} \mid \mathtt{L}) &= \mathcal{H}(\mathtt{O}, \mathtt{L}) - \mathcal{H}(\mathtt{L}) \\
&= \mathcal{H}(f_0(f_1(\mathtt{L}), \mathtt{H} \underline{\oplus} f_2(\mathtt{L})), \mathtt{L}) - \mathcal{H}(\mathtt{L}) \\
&\leq \mathcal{H}(f_1(\mathtt{L}), \mathtt{H} \underline{\oplus} f_2(\mathtt{L}), \mathtt{L}) - \mathcal{H}(\mathtt{L}) && \text{(by } \mathcal{H}(f(X)) \leq \mathcal{H}(X)\text{)} \\
&= \mathcal{H}(\mathtt{H} \underline{\oplus} f_2(\mathtt{L}), \mathtt{L}) - \mathcal{H}(\mathtt{L}) && \text{(by the definition of } \mathcal{H}\text{)} \\
&= \mathcal{H}(\mathtt{H} \underline{\oplus} f_2(\mathtt{L}) \mid \mathtt{L}) && \text{(by the definition of } \mathcal{H}(\mathtt{X} \mid \mathtt{Y})\text{)} \\
&\leq \mathcal{H}(\mathtt{H} \underline{\oplus} f_2(\mathtt{L}) \mid f_2(\mathtt{L}))
\end{aligned}$$

Thus, $\mathcal{I}(\mathtt{O}; \mathtt{H} \mid \mathtt{L})$ is bound by the maximum information leakage by the operation $\underline{\oplus}$ (more precisely, the maximum value of $\mathcal{H}(\mathtt{H} \underline{\oplus} \mathtt{X} \mid \mathtt{X})$ obtained by changing the distribution for $\mathtt{X}$).

If $\underline{\oplus}$ is the equality test for k-bit integers, then

$$\begin{aligned}
\mathcal{H}(\mathtt{H} \underline{\oplus} \mathtt{X} \mid \mathtt{X}) &= P(\mathtt{H} = \mathtt{X}) \log \tfrac{1}{P(\mathtt{H}=\mathtt{X})} + P(\mathtt{H} \neq \mathtt{X}) \log \tfrac{1}{P(\mathtt{H}\neq\mathtt{X})} \\
&= \tfrac{1}{2^k} \log 2^k + \tfrac{2^k-1}{2^k} \log \tfrac{2^k}{2^k-1} \;\leq\; \tfrac{k+1}{2^k}
\end{aligned}$$

Thus, the maximum leakage is bound by $\frac{k+1}{2^k}$ (which is considered safe if k is sufficiently large).

On the other hand, if $\underline{\oplus}$ is the inequality test $<$, then, the maximum value of $\mathcal{H}(\mathtt{H} \underline{\oplus} \mathtt{X} \mid \mathtt{X})$ is obtained by letting $P(\mathtt{X} = 2^{k-1}) = 1$.

$$\mathcal{H}(\mathtt{H} \underline{\oplus} \mathtt{X} \mid \mathtt{X}) = P(\mathtt{H} < 2^{k-1}) \log \tfrac{1}{P(\mathtt{H}<2^{k-1})} + P(\mathtt{H} \geq 2^{k-1}) \log \tfrac{1}{P(\mathtt{H}\geq 2^{k-1})} = 1$$

Thus, we know that 1 bit of information about σ may be leaked by each run of the program.

Note that the above discussion, we used only the fact that $\langle \Sigma, D, e\rangle$ satisfies linear relaxed NI; the discussion applies to *any* program e that satisfies the policy Σ and D. Thus, the quantity of information leakage can be estimated only by looking at Σ and D.

[5] Note that we are considering deterministic programs. Note also that we do not consider timing attacks. It is possible to hide timing attacks to some extent, by using Agat's technique, for instance [1].

5 Related Work

There have been many studies on information flow security and declassification policies: see [11,13] for a general survey and comparison of declassification policies. Most closely related to our work is Sabelfeld and Myers' work on delimited information release [12], and Li and Zdancewic's work on relaxed NI [7]. They control *what* functions can be used for declassification, but not *how often* the declassification functions may be used. Controlling *what* declassification functions are used is sufficient if the declassification functions do not return functions. In fact, in delimited information release, one can use $\lambda x.(l = x)$ (where l is a low security variable) for the password example; No matter how often declassification is performed, the leaked information is the one bit information $h = l$. (In the relaxed NI [7], this is not allowed since policies must be closed terms.) If the declassification functions return functions (as in the password example in this paper), however, controlling *what* declassification functions are used is not sufficient for bounding the quantity of information leakage. In the case of the password example, if one wants to specify that the password can be compared with *some* string but does not to want to specify which string should be compared with the password, then one should use $\lambda x.\lambda s.(s = x)$ as the declassification function. We should therefore control *how often* functions are used to bound the quantity of information leakage.

Another approach to extending relaxed NI would be to replace the equivalence relation in the definition of relaxed NI with a complexity-preserving relation, as discussed in [13]. Let us write $e \succeq e'$ if e' is more efficient than e (see [14] for formal discussion of such a relation). Then, if $e \succeq e'(d\,h)$ holds for some e' that does not contain h, e cannot declassify information about h much faster than by calling the declassification function d. In the password example (where $d = \lambda x.\lambda s.(x = s)$), $e \succeq e'(d\,h)$ implies that it takes a time exponential in the bit length of h for e to leak the entire information about h. Thus, this approach is useful for estimating the speed of information leakage. The approach, however, sometimes gives too conservative estimation of the rate of information leakage. For example, PIN code for a bank account typically consists of only 4 digits, hence knowing that a program satisfies the complexity-preserving relaxed NI for the declassification function $\lambda x.\lambda s.(x = s)$ does not give enough security assurance (because calling the declassification function 10^4 times would not take a second). On the other hand, if the program satisfies the linear relaxed NI, the PIN code can be tested only once per program run, so that we can obtain reasonable security assurance by controlling how often the program can be run.

Li and Zdancewic's type system for relaxed NI [7] allows more flexible declassification than ours; for example, if a declassification function for σ is $\lambda x.((x+1) = 2)$, then declassification can be performed in two steps, by first applying $\lambda x.x+1$ and then $\lambda y.y = 2$. We think it is possible to extend our linear type system to allow such flexible declassification.

Quantitative analysis of information flow has been recently studied by Malacaria et al. [8,3,2] for imperative languages. As demonstrated in Section 4.3, the linear relaxed non-interference allows us to apply quantitative analysis only

to declassification functions instead of the whole program, by which enabling a combination of traditional information flow analysis (with linearity analysis) and quantitative information flow analysis. A limitation of our approach is that only $0, 1, \omega$ uses are considered, so that if a declassification is performed inside a recursive function, the number of declassifications is always estimated as ω. To remove that limitation, we need to generalize uses, possibly using dependent types (for example, we can write $\Pi n : int_{\mathbf{L}}.int_{\{d \mapsto n\}} \to int_{\mathbf{L}}$ for the type of functions that takes an integer n and a high-security value x, and applies the declassification function d to x, n times).

Our type system can be considered an instance of linear type systems [17,6,9]. In the usual linear type systems, the type of an integer is annotated with how often the integer is accessed. In our type system, the type of an integer is annotated with how often each declassification function may be applied to the integer. We did not discuss a type inference algorithm in this paper, but a type inference algorithm (that is quadratic in the program size, provided that the number of declassification functions is constant) can be developed in a standard manner [9].

6 Conclusion

We introduced a new notion of declassification called *linear declassification*, which not only controls what functions can be used for declassifying high-security values but also *how often* the declassification functions may be applied. We have also introduced *linear relaxed non-interference* to formalize the property guaranteed by linear declassification. The linear relaxed non-interference enables integration of traditional type-based information flow analysis and quantitative information flow analysis, by allowing us to apply quantitative analysis locally to declassification functions.

In the paper, we used password checking as the motivating example. It is left for future work to study more applications of linear declassification. We used a static type system to guarantee linear relaxed NI. Combining our approach with dynamic analysis (for counting of how often functions are called) would also be an interesting direction for future work.

Acknowledgment. We would like to thank Andrei Sabelfeld and Eijiro Sumii for discussions on this work. We would also like to thank anonymous referees for useful comments.

References

1. Agat, J.: Transforming out timing leaks. In: Proc. of POPL, pp. 40–53 (2000)
2. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation 15(2), 181–199 (2005)
3. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15(3), 321–371 (2007)

4. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
5. Kaneko, Y., Kobayashi, N.: Linear declassification (2007), `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/lindcl-full.pdf` (extended version)
6. Kobayashi, N.: Quasi-linear types. In: Proc. of POPL, pp. 29–42 (1999)
7. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proc. of POPL, pp. 158–170 (2005)
8. Malacaria, P.: Assessing security threats of looping constructs. In: Proc. of POPL, pp. 225–235 (2007)
9. Mogensen, T.: Types for 0, 1 or Many Uses. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 112–122. Springer, Heidelberg (1998)
10. Pottier, F., Simonet, V.: Information flow inference for ML. In: Proc. of POPL, pp. 319–330 (2002)
11. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
12. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
13. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security (to appear). A preliminary version appeared in Proceedings of 18th IEEE Computer Security Foundations Workshop (CSFW-18), pp. 255-269 (2005)
14. Sands, D., Gustavsson, J., Moran, A.: Lambda Calculi and Linear Speedups. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 60–82. Springer, Heidelberg (2002)
15. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal 27, 379–423 (1948)
16. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proc. of POPL, pp. 355–364 (1998)
17. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: Proceedings of Functional Programming Languages and Computer Architecture, pp. 1–11. San Diego, California (1995)