Unifying Confidentiality and Integrity in Downgrading Policies

Peng Li Steve Zdancewic
Department of Computer and Information Science
University of Pennsylvania
{lipeng,stevez}@cis.upenn.edu

May 13, 2005

Abstract

Confidentiality and integrity are often treated as dual properties in formal models of information-flow control, access control and many other areas in computer security. However, in contrast to confidentiality policies, integrity policies are less formally studied in the information-flow control literature. One important reason is that traditional noninterference-based information-flow control approaches give very weak integrity guarantees for untrusted code. Integrity and confidentiality policies are also different with respect to implicit information channels.

This paper studies integrity downgrading policies in information-flow control and compares them with their confidentiality counterparts. We examine the drawbacks of integrity policies based on noninterference formalizations and study the integrity policies in the framework of downgrading policies and program equivalences. We give semantic interpretations for traditional security levels for integrity, namely, tainted and untainted, and explain the interesting relations between confidentiality and integrity in this framework.

Keywords: language-based security, information flow, integrity, downgrading, security policy.

1 Introduction

Language-based information-flow security [11] provides end-to-end guarantees on the dependency and the propagation of information in the system, which is usually formalized as *noninterference* [2, 5] properties. Such security guarantees are ideal for protecting *confidentiality*, where secret information is not permitted to propagate to public places. On the other hand, information-flow control can also be used to provide *integrity* guarantees, where important data in the system is not allowed to be affected by untrusted sources of information. Confidentiality and integrity can be viewed as duals [1] in many areas of computer security. In information flow, confidentiality policies prevent secret data from being leaked out to the adversary, while integrity policies restrict the use of data coming from the adversary.

There are very practical applications of information-flow policies for *integrity*. For example, an unsafe script on the web server could use strings from untrusted inputs to compose a SQL query string and then have the database management system execute the query, which potentially allow the attacker to execute commands in the database. The Perl programming language provides built-in support for dynamic information-flow checking. Data from user inputs and the network is marked as tainted, while system calls require untainted data. Tainted data can be converted to untainted data through pattern matching, which effectively forces the programmer to examine untrusted data and avoid malicious attacks. Code analysis tools such as cqual [12] perform static information-flow checking to detect the use of dangerous data. Such tools have been used to find bugs in large-scale software systems.

Despite the practical interests, integrity policies are often less formally studied in the literature of language-based information-flow security. Many formal studies are focused on confidentiality and merely mention that confidentiality and integrity are duals. In fact, confidentiality and integrity are not symmetric in traditional approaches based on noninterference. The noninterference property is too emphasized for confidentiality and it is not appropriate for integrity. For example, noninterference does not give useful integrity guarantees for untrusted code, and it is often too strong for practical use because it rules out all implicit information flows, most of which are not harmful. Noninterference also does not handle downgrading. Section 2 identifies these challenges on integrity policies in information-flow.

To fix the aforementioned weaknesses of noninterference-based integrity policies, we need alternative formalizations for information flow. Out recent research [4] uses downgrading policies and program equivalences to formalize the goals of language-based information-flow security. The original motivation is to achieve an end-to-end security guarantee like noninterference when *downgrading* (or declassification) is available in the system. While this framework was originally focused on confidentiality, it also provides a basis for very expressive integrity downgrading policies. Moreover, it is possible to achieve a better formal security goal for integrity that avoids the drawbacks of noninterference-based definitions. Section 3 extends the framework of downgrading policies with integrity policies. We discuss how to formalize the security goal for integrity present a highly symmetric view over confidentiality and integrity in this framework.

2 Challenges of Integrity

2.1 Policy expressiveness

The formal definition of noninterference gives an intuitive and absolute meaning of confidentiality, but its relationship with integrity is less straightforward. In general, integrity has many meanings in computer security. For example, Pfleeger's security textbook [8] describes integrity policies that require that: data is modified only by authorized principals, data is modified in permitted ways, data is consistent, valid, meaningful and correct, etc. The actual meaning of integrity depends on the specific context. Noninterference only provides a particular kind of integrity guarantee, that is, trusted data is not affected by the propagation of untrusted data. Apparently, there are many information integrity policies that noninterference cannot express. Most useful integrity policies involve accurate description of the actual computation. Integrity policies should not only specify who modified the data, but also specify how the data is manipulated.

2.2 Untrusted code

For untrusted code, noninterference gives a strong and practically useful guarantee for data confidentiality. This makes information-flow control a killer application for safely executing untrusted programs while giving them accesses to secret information.

However, traditional noninterference gives almost no integrity guarantee for untrusted code. The reason is that, when the code is not trusted, the adversary can manipulate trusted data in arbitrary ways in the program. For example, suppose the following function `foo` is written by an adversary. It takes two input arguments, performs some computation and returns a `untainted` value.

```
untainted int foo( untainted int a, tainted int b) { return a-a+0xff00; }
```

Although `foo` satisfies the noninterference policy, i.e. there is no information flow from the `tainted` input `b` to the `untainted` result, the result is not at all trustworthy because the adversary can return any arbitrary value in this function. Therefore, the data coming from untrusted programs (or software modules) must always be treated as tainted. Integrity policies based on noninterference definitions can only used in trusted environments, where the programmers are cooperative and goal is to prevent accidental security exploits in trusted code. For the same reason, the two-dimensional *decentralized label model* [7] for confidentiality falls back to a one-dimensional model for integrity [3], which makes the integrity labels much less expressive in languages with information-flow type systems, such as FlowCaml [13, 9, 10] and Jif [6].

2.3 Downgrading

Pure noninterference is too ideal for practical applications. Most of the time, we do need to use information from untrusted sources in trusted places, as long as the tainted data can are verified to be safe. In the example of taint-checking mode in Perl, tainted data can be converted to untainted using pattern-matching. Clearly, there can be information propagation from untainted data to tainted, and noninterference policies are not directly applicable. This is the dual case for confidentiality, where secret data also needs to be declassified. This paper extends the framework of *downgrading policies* [4] and presents a symmetrical version of integrity downgrading policies, sometimes called *endorsement*.

2.4 Implicit information flow

Most confidentiality policies do not tolerate implicit information leakage. There are many implicit information channels such as control flow, timing channels and various side-effects that must be considered when untrusted code is available. For example, the following code has an implicit information leak from `secret` to `x` via control flow. If the code is not trusted and `x` is a publicly visible, the adversary can easily know the last bit of `secret` by observing the value of `x`. Noninterference policies rules out such implicit flows.

```
if (secret%2=1) then x:=1 else x:=0;
```

A straightforward solution is to use *downgrading* on the branching conditions where implicit flows are allowed. However, such implicit information propagation is almost always acceptable for data integrity policies, where the programmers are trusted and the goal is to prevent accidental destruction of trustworthy data. For example, the taint-checking mode in Perl does not check implicit information flows at all. Since the value of any trustworthy data can be directly modified by the programmer without violating information-flow policies as we have shown in the `foo` function above, there are few reasons to prevent implicit information flows, which are much more difficult to exploit to cause damage. Therefore, the security policy for protecting integrity does not have to be as strict as pure noninterference policies. Implicit information flow should be allowed by default, without the awkwardness of using explicit downgrading mechanisms.

3 Downgrading Policies for Integrity

To avoid the drawbacks of noninterference-based integrity policies, we study them in an alternative formal framework. Our recent research [4] uses downgrading policies and program equivalences to formalize the security goals of language-based information-flow security. This framework was originally focused on confidentiality, and this paper extends the integrity aspect of it. Similar to confidentiality labels, we define a partial ordering on integrity labels, formalizes the downgrading relation for integrity, and give interpretations to traditional security levels such as *untainted* and *tainted*. To highlight the symmetry between confidentiality and integrity, the definitions for two kinds of policies are given in parallel for the rest of the paper.

Briefly, this framework uses *downgrading policies* to express security levels of data and define the ordering among these security levels, which generalizes the simple security lattices $\mathsf{public} \sqsubseteq \mathsf{secret}$ for confidentiality and $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$ for integrity. A security level is simply a non-empty set of downgrading policies, where each policy describes the computation related to downgrading. We reason about the programs in an end-to-end fashion. Each program takes input data and produces output data. Confidentiality policies are specified for each program input; integrity policies are specified for the program output. The security goal is then formalized using such security policies.

3.1 Downgrading policies and security labels

type	$\tau ::=$	$\mathsf{int} \mid \tau \rightarrow \tau$
constant	$c ::=$	$0, 1, 2, ...$
operator	$\oplus ::=$	$+, -, \%, =, ...$
downgrading policy	$m ::=$	$\lambda x : \tau . m \mid m\ m \mid x \mid c \mid\ \oplus\ \mid \mathsf{if}\ m\ m\ m$
confidentiality label	$cl ::=$	$\{m_1, \ldots, m_k\} \mid \mathsf{secret}_\tau \mid \mathsf{public}_\tau$
integrity label	$il ::=$	$\{m_1, \ldots, m_k\} \mid \mathsf{tainted}_\tau \mid \mathsf{untainted}_\tau$

Figure 1: Downgrading Policies and Security Labels

The syntax of downgrading policies and security labels is shown in Figure 1. Each downgrading policy is a term in the simply-typed λ-calculus, extended with operators and constants. The policy language is intended to be a fragment of the full language for which equivalence is decidable, so that the primitive operators will match those in the full language. Each downgrading policy represents some computation associated with downgrading as following:

- Confidentiality policies: each policy is a function that specifies how the data can be released to the public in the future. When this function is applied to the annotated data, the result is considered as public. For example, if a secret variable x is annotated with the confidentiality policy $\lambda x. x\%2$, it means the last bit of x can be released to public.

- Integrity policies: each policy is a term that specifies how the data has been computed in the past. For example, the integrity policy "2" means the data must be equal to 2 and works like a singleton type. The policy can be a function, too. For example, the policy $\lambda x. x\%10$ for an integer means that the integer must have been computed by $x\%10$, where x is potentially untrusted and we do not know what x is. Another useful policy is $\lambda x. \lambda y. \mathsf{match}(x, y)$, where match is a predefined pattern matching function and the policy means the data is the result of pattern matching. The weakest policy is the identity function, $\lambda x. x$, which simply gives no information about the how the data has been computed in the past.

Policies are typed in the simply-typed λ-calculus using the judgment $\Gamma \vdash m : \tau$. We use standard β-η equivalences $\Gamma \vdash m_1 \equiv m_2 : \tau$ for policy terms. Policy terms can be composed as functions using the following definitions.

Definition 3.1.1 (Policy composition) *If $\Gamma \vdash m_1 : \tau_1 \rightarrow \tau_3$ and $\Gamma \vdash m_2 : \tau_2 \rightarrow \tau_1$, the composition of m_1 and m_2 is defined as $m_1 \circ_\Gamma m_2 \stackrel{\triangle}{=} \lambda x : \tau_2 . m_1\ (m_2\ x)$*

Definition 3.1.2 (Multi composition) *If $\Gamma \vdash m_1 : \tau \rightarrow \tau'$ and $\Gamma \vdash m_2 : \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \tau$, the multi-composition of m_1 and m_2 is defined as $m_1 \odot_\Gamma m_2 \stackrel{\triangle}{=} \lambda x_1 : \tau_1 . \ldots \lambda x_k : \tau_k . m_1 (m_2\ x_1 \ldots x_k)$*

Given the definition of downgrading policies, we can define a security label as a non-empty set of downgrading policies. We slightly abuse the notation to use cl to range over confidentiality labels and il to range over integrity labels. Labels are well-formed with respect to the type of data it annotates.

Definition 3.1.3 (Label wellformedness)
$\boxed{\vdash cl \lhd \tau} \iff \forall m \in cl, \exists \tau_1, \vdash m : (\tau \to \tau_1)$
$\boxed{\vdash il \lhd \tau} \iff \forall m \in il, \exists \tau_1, \ldots, \exists \tau_k, \vdash m : (\tau_1 \to \ldots \to \tau_k \to \tau)$

The meanings of two kinds of labels are symmetric:

- Confidentiality labels: each policy in the label can be used to declassify the data in the future. For example, if the value p is annotated with the confidentiality label $\{(\lambda p.\, \lambda x.\, p{=}x),\ (\lambda p.\, p\%2)\}$, it means that p can be declassified by comparing it with some other value, or by extracting its last bit.
- Integrity labels: each policy describes a possible computation that generated the value as the result in the past. For example, if the value p is annotated with the integrity label $\{(\lambda x.\, \mathsf{match}(x, c_1)), c_2\}$, it means the value is either the result of pattern-matching against pattern c_1 or a predefined constant c_2.

3.2 Label Ordering

Each label is syntactically represented as a set of downgrading policies, but the semantics of the label includes far more policies than explicitly specified. We define the interpretation of security labels as the following.

Definition 3.2.1 (Label interpretation)
$\mathbb{S}_\tau(cl) \triangleq \{n' \mid n \in cl, \quad \vdash n' \equiv m \circ_\Gamma n : \tau_2\}$
$\mathbb{S}_\tau(il) \triangleq \{n' \mid n \in il, \quad \vdash n : (\tau_1 \to \ldots \to \tau_k \to \tau),$
$\qquad \vdash n' \equiv (\lambda x_1 : \tau_1'. \ldots \lambda x_i : \tau_i'.\, n\, m_1 \ldots m_j) : (\tau_1' \to \ldots \to \tau_i' \to \tau)\ \}$

To understand the above definitions, suppose the security label has the policy n:

- For confidentiality: any function m composed with n is also a valid downgrading policy implied by n. The intuition is that if $(n\ x)$ is public, then $(m\ (n\ x))$ is also public, no matter what m is.
- For integrity: if we can compose n with some other terms and get a larger term n', in which n represent the final step of computation, then n' is also implied in this label, because any data computed by n' can also be treated as if it is computed by n using some input data.

  For example, suppose we have an integrity label $il_1 \triangleq \{\lambda x.\, \mathsf{match}(x, c_1)\}$, then the following policies are also in $\mathbb{S}(il_1)$: $(\lambda a.\, \lambda b.\, \mathsf{match}(a + b, c_1))$, $(\mathsf{match}(c_2, c_1))$, etc. Intuitively speaking, if we only know that the data is the result of pattern-matching against the pattern c_1, then there are possibilities that the value matched with c_1 could be $a + b$, c_2, or any other values.

Based on the semantics of labels, we can easily define the ordering on security labels, using the set inclusion relation on label interpretations. We use the notation $l_1 \sqsubseteq l_2$ to say l_2 is a higher security level than l_1. The definitions for confidentiality and integrity labels are completely symmetric.

Definition 3.2.2 (Label ordering)
$\boxed{cl_1 \sqsubseteq cl_2 \lhd \tau} \iff \mathbb{S}_\tau(cl_1) \supseteq \mathbb{S}_\tau(cl_2), \quad \boxed{il_1 \sqsubseteq il_2 \lhd \tau} \iff \mathbb{S}_\tau(il_1) \subseteq \mathbb{S}_\tau(il_2).$

- Confidentiality: high security levels correspond to secret levels, low security levels corresponds to public levels. The intuition is that, each policy in the label corresponds to a path where secret data can be released to public places. The fewer paths there are, the more secure the data is.

- Integrity: high security levels correspond to tainted or untrusted levels and low security levels corresponds to trusted levels, because we would like to allow information flow from low levels to high levels but not in the other direction. The intuition is that, each policy corresponds to a possible computation that have generated the data. The more possibilities there are, the less trustworthy the data is.

We claim that the ordering of security labels generalizes the two-point lattices $\mathsf{public} \sqsubseteq \mathsf{secret}$ and $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$. In fact, all these traditional security levels can be interpreted in the framework of downgrading policies:

Definition 3.2.3 (Interpretation of special labels)
$\mathsf{secret}_\tau \triangleq \{\lambda x\!:\!\tau.0\}$ $\mathsf{public}_\tau \triangleq \{\lambda x\!:\!\tau.x\}$ $\mathsf{tainted}_\tau \triangleq \{\lambda x\!:\!\tau.x\}$ $\mathsf{untainted}_\tau \triangleq \{m \mid\ \vdash m\!:\!\tau\}$

- Confidentiality: we can prove that all the confidentiality labels of a given type form a lattice, where secret_τ is the top and public_τ is the bottom.

- Integrity: we can prove that $\mathsf{tainted}_\tau$ is the highest label of all the integrity labels. However, there is no single lowest label in the integrity ordering. Instead, there are many different lowest labels. For example, suppose $\tau = \mathsf{int}$, then $\{c_0\}$, $\{c_1\}$ and so on are all lowest labels. For a set of labels, their join (least upper bound) always exists, but they may not have a lower bound.

  The interpretation of $\mathsf{untainted}_\tau$ is the set of policies representing computations that do not use potentially untrusted inputs. We choose this interpretation for two reasons. First, it reflects the meaning of "untainted", i.e. the corresponding computation did not use tainted data. Second, its semantics is backward compatible with untainted in the traditional two-point security lattice, because when two untainted data meets together during computation (i.e. when computing $c_1 + c_2$), the result can also be labeled as untainted, which is the way things work in the two-point lattice. Thus, the ordering on integrity labels can be understood as a refined version of the two-point lattice $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$.

  Apparently, $\mathsf{untainted}_\tau$ does not provide a very strong security guarantee: data with this label can have any value. This coincides with the facts we mentioned in Section 2.2: if the code is not trusted, then untainted data can be anything. However, we now have security levels that provide more precise security guarantees: data with label $\{(\lambda x.\mathsf{match}(x, c_1))\}$ are guaranteed to match the pattern c_1; data with label $\{c_1, c_2\}$ is either c_1 or c_2, etc. Interestingly, the label $\{(\lambda x.\mathsf{match}(x, c_1))\}$ is not lower than untainted, but it provides a much more precise security guarantee than untainted does.

Overall, we can see that the label orderings for confidentiality and integrity are highly symmetric. The security levels public and tainted are both represented using the identity function and they both refer to data under control of the attacker. The security levels secret and untainted are also symmetric in some sense: secret are represented using constant functions, while untainted is represented using a set of terms that can be statically evaluated to normal forms.

The only asymmetry is that there are multiple lowest integrity labels, while there is only one highest confidentiality label. In fact, each lowest integrity label $\{c\}$ has its counterpart $\{\lambda x.\,c\}$ in the confidentiality lattice. It is just that all confidentiality labels $\{\lambda x.\,m\}$ such that x is free in m are structurally equivalent because their label interpretation are the same as the interpretation of a constant function. Intuitively speaking, different constant policies provide different integrity guarantees, but all constant policies have the same effect for confidentiality. This fact, together with the thoughts in Section 2.1, show a important difference between confidentiality and integrity in information flow. Confidentiality policies are destructive and do not care about the actual computation of secret data. If the secret data is destroyed and becomes garbage, it does not violate any confidentiality policies and the system is still secure. In contrast, integrity policies are highly related to the correctness and precision actual computation performed on the data.

3.3 Label Downgrading

The security label of data changes as the data is involved in some computation. We use the concept of *label downgrading* to describe the transition of security labels. To formalize this concept, suppose the data x_1 has type τ_1, and it is annotated by labels cl_1 and il_1. We use the concept of an *action* to model the computation on x_1: an action m on x_1 is a function applied to x_1. For example, suppose the computation on x_1 is $\mathsf{hash}(x_1)$, then the action is simply the hash function. If the computation is $x_1 + y$, then the action is $(\lambda x. x + y)$. Now, given the action m, suppose the result $(m\ x_1)$ has type τ_2, we can formally define the label cl_2 and il_2 on the result:

Definition 3.3.1 (Label Downgrading)

$$\boxed{(cl_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (cl_2 \lhd \tau_2)} \iff \vdash cl_1 \lhd \tau_1,\ \ \vdash cl_2 \lhd \tau_2,\ \ \forall m_2 \in cl_2, \exists m_1 \in \mathbb{S}_{\tau_1}(cl_1),\ \ \vdash m_2 \odot_\Gamma m \equiv m_1 : \tau$$

$$\boxed{(il_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (il_2 \lhd \tau_2)} \iff \vdash il_1 \lhd \tau_1,\ \ \vdash il_2 \lhd \tau_2,\ \ \forall m_2 \in il_2, \exists m_1 \in \mathbb{S}_{\tau_1}(il_1),\ \ \vdash m_2 \equiv m \odot_\Gamma m_1 : \tau$$

The judgment $(cl_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (cl_2 \lhd \tau_2)$ can be read as "the confidentiality label cl_1 at type τ_1 is transformed to a label cl_2 at type τ_2 under the action m". The definition is completely symmetric for confidentiality labels and integrity labels. Let us understand these rules by looking at some examples. For simplicity and readability, we omit the typing information in the following examples.

- Confidentiality labels: suppose we have the following labels and actions that are related using the label downgrading definition.

$$\begin{array}{lll} cl_1 \triangleq \{\lambda x.\lambda y.\mathsf{hash}(x)\%4 = y\} & m_1 \triangleq \lambda x.\mathsf{hash}(x) & (cl_1 \lhd \mathsf{int}) \overset{m_1}{\rightsquigarrow} (cl_2 \lhd \mathsf{int}) \\ cl_2 \triangleq \{\lambda x.\lambda y.x\%4 = y\} & m_2 \triangleq \lambda x.x\%4 & (cl_2 \lhd \mathsf{int}) \overset{m_2}{\rightsquigarrow} (cl_3 \lhd \mathsf{int}) \\ cl_3 \triangleq \{\lambda x.\lambda y.x = y\} & m_3 \triangleq \lambda x.\lambda y.x = y & (cl_3 \lhd \mathsf{int}) \overset{m_3}{\rightsquigarrow} (\mathsf{public}_{\mathsf{int}} \lhd \mathsf{int}) \end{array}$$

  Suppose $x_2 \triangleq \mathsf{hash}(x_1)$, $x_3 \triangleq x_2\%4$ and $x_4 \triangleq (x_3 = p)$. If x_1 has label cl_1, then x_2 has label cl_2, x_3 has label cl_3, x_4 has label $\mathsf{public}_{\mathsf{int}}$. Intuitively speaking, the downgrading policies in a confidentiality label describe paths in which data can be downgraded in the future, which may involve several steps of computation. In the downgrading relation, the action m matches the prefix of such a path m_1, and the remaining path m_2 is preserved in the resulting label.

- Integrity labels: suppose we have the following labels, actions and downgrading relations:

$$\begin{array}{lll} il_1 \triangleq \mathsf{tainted}_{\mathsf{int}} & m_1 \triangleq \lambda x.\mathsf{match}(x, c_1) & (il_1 \lhd \mathsf{int}) \overset{m_1}{\rightsquigarrow} (il_2 \lhd \mathsf{int}) \\ il_2 \triangleq \{\lambda x.\mathsf{match}(x, c_1)\} & m_2 \triangleq \lambda x.x + c_2 & (il_2 \lhd \mathsf{int}) \overset{m_2}{\rightsquigarrow} (il_3 \lhd \mathsf{int}) \\ il_3 \triangleq \{\lambda x.\mathsf{match}(x, c_1) + c_2\} & & \end{array}$$

  Suppose $x_2 \triangleq \mathsf{match}(x_1, c_1)$ and $x_3 \triangleq x_2 + c_2$. If x_1 has label $\mathsf{tainted}_{\mathsf{int}}$, then x_2 has label il_2 and x_3 has label il_3. Intuitively speaking, the downgrading policies in an integrity label approximate the computation in the past, from which the data could have been computed. In the downgrading relation, the action m is appended to the history of computation m_1, and the result m_2 is in the resulting label.

3.4 Security Goals

The main question is: how to tell that a program is safe with respect to some security policies? We formalize the security goal in a language based on the simply-typed lambda calculus, which is basically an extension of our policy language. We define the security goals as end-to-end properties on the input-output relationship of the program.

Rather than using operational semantics, we use static program equivalences in the definition: if the program is safe, it must be equivalent to some special forms. The equivalence relation $\equiv$ is the standard $\beta - \eta$ equivalence, extended with some trivial rules for conditional expressions such as $e_1 \equiv \mathsf{if}\ 1\ e_1\ e_2$. The full definition of the equivalence relation is similar to those in our earlier work [4].

Definition 3.4.1 (Relaxed Noninterference) *Suppose the program uses secret input variables $\sigma_1, \sigma_2, \ldots$, where each input variable σ_i has a confidentiality label $\Sigma(\sigma_i)$ specified by the end user. For a program output e at type τ:*

- *e satisfies the confidentiality policy Σ, if $e \equiv f\ (m_1\ \sigma_{a_1}) \ldots (m_n\ \sigma_{a_n})$, where $\forall i.\sigma_i \notin FV(f)$ and $\forall j.m_j \in \Sigma(\sigma_{a_j})$.*
- *e satisfies the integrity policy il, if $e \equiv \begin{pmatrix} \mathsf{if}\ e_1 (m_1\ e_{11}\ e_{12}\ ...) \\ \mathsf{if}\ e_2 (m_2\ e_{21}\ e_{22}\ ...) \\ .... \\ \mathsf{if}\ e_n (m_n\ e_{n1}\ e_{n2}\ ...) \\ \quad (m_0\ e_{01}\ e_{02}\ ...) \end{pmatrix}$ where $\forall j.m_j \in il$.*

The confidentiality condition requires that the program can be rewritten to a special form where secret variables are leaked to public places by using only the permitted functions (downgrading policies). Confidentiality policies are specified on the *input* of the program. The integrity condition requires that the program can be rewritten to a special form where the result is computed using one of the functions (downgrading policies) in the integrity label. Integrity policies are specified on the *output* of the program.

To understand the integrity guarantee better, consider the following two possibilities:

1. There is only one policy m in il. In this case, all the branches have the same m, so we have the following equivalence:
$$\begin{pmatrix} \mathsf{if}\ e_1 (m\ e_{11}\ e_{12}\ ...) \\ \mathsf{if}\ e_2 (m\ e_{21}\ e_{22}\ ...) \\ .... \\ \mathsf{if}\ e_n (m\ e_{n1}\ e_{n2}\ ...) \\ \quad (m\ e_{01}\ e_{02}\ ...) \end{pmatrix} \equiv m \begin{pmatrix} \mathsf{if}\ e_1 (e_{11}) \\ \mathsf{if}\ e_2 (e_{21}) \\ .... \\ \mathsf{if}\ e_n (e_{n1}) \\ \quad (e_{01}) \end{pmatrix} \begin{pmatrix} \mathsf{if}\ e_1 (e_{12}) \\ \mathsf{if}\ e_2 (e_{22}) \\ .... \\ \mathsf{if}\ e_n (e_{n2}) \\ \quad (e_{02}) \end{pmatrix} ...$$
This provides a very straightforward security guarantee. The attacker can only affect the result by *downgrading*, i.e. let untrusted data go through the downgrading policy m.

2. There are multiple policies in il. The body of each branch is still protected by a downgrading policy in il, but the attacker also has the ability to choose the exact branch to be taken, thus affecting the result via implicit control flow. Such implicit flows cannot be easily justified by *downgrading*, because the conditions $e_1 ... e_n$ are arbitrary programs not related to the downgrading policy. Our definition simply permits such implicit flows because we defined the integrity label as a set of *possible* computations. No matter which branch is taken, the final step of computation is always captured by the integrity label. This definition meets our requirement in Section 2.4. In contrast to the integrity condition, the confidentiality condition of Definition 3.4.1 does not tolerate any implicit information flow.

Definition 3.3.1 shows the symmetry between confidentiality and integrity conditions in a simple way: it describes how secret data are leaked to public places (for confidentiality) and how untrusted program generates trusted result (for integrity). However, it only specify policies on one side of the program and assumes that the program output is public and that the program inputs are tainted. Definition 3.3.1 can be further generalized to achieve more fine-grained end-to-end security conditions.

Definition 3.4.2 (Relaxed Noninterference (refined)) *Suppose the program uses input variables $\sigma_1, \sigma_2, \ldots$, where each input variable σ_i has a confidentiality label $\Sigma_c(\sigma_i)$ and integrity label $\Sigma_i(\sigma_i)$ specified by the end user. For a program output e at type τ:*

- *e satisfies the confidentiality policy cl, if $\forall n \in cl$, $(n\ e) \equiv f\ (m_1\ \sigma_{a_1}) \ldots (m_n\ \sigma_{a_n})$, where $\forall i.\sigma_i \notin FV(f)$ and $\forall j.m_j \in \Sigma_c(\sigma_{a_j})$.*
- *e satisfies the integrity policy il, if $\forall n_1 \in \Sigma_i(\sigma_1)$, ... $\forall n_k \in \Sigma_i(\sigma_k)$, for all f_{xy} that make the following substitutions well-typed,*

$$[(n_1\ f_{11}\ f_{12}\ ...)/\sigma_1]\ ...\ [(n_k\ f_{k1}\ f_{k2}\ ...)/\sigma_k]\ e \equiv \begin{pmatrix} \text{if } e_1(m_1\ e_{11}\ e_{12}\ ...) \\ \text{if } e_2(m_2\ e_{21}\ e_{22}\ ...) \\ .... \\ \text{if } e_n(m_n\ e_{n1}\ e_{n2}\ ...) \\ \quad (m_0\ e_{01}\ e_{02}\ ...) \end{pmatrix}$$

where $\forall i.\sigma_i \notin FV(f_{xy})$ and $\forall j.m_j \in il$.

In Definition 3.4.2, security policies are uniformly specified on both ends the program: Σ_c, Σ_i specify policies on the program inputs and cl, il specify policies on the program output. The confidentiality condition allows the program output to have security levels other than public. Compared to the integrity condition in Definition 3.4.1 where the program is simply untrusted, Definition 3.4.2 allows us to give trusted data to an untrusted program yet still having guarantees on the program output. The integrity condition looks more verbose because we have to use a lot of variables and term substitutions. However, the confidentiality condition and integrity condition are inherently symmetric except that the integrity condition allows implicit flows (via the if expressions).

3.5 Extensions

Similar to the idea of *global* downgrading policies in our previous framework [4], we can extend the policy language with secret variables. Although this significantly changes the confidentiality lattice (for example, the policy public is no longer the bottom of the lattice), the ordering of integrity labels is largely unchanged. In fact, doing so will only make the integrity policies more expressive. The integrity labels $\{\sigma_1\}$ and $\{c_1\}$ are very much alike — they are both singleton types; they are all lowest labels in the integrity ordering.

4 Conclusion

This paper studies the challenges on integrity policies in language-based information-flow security and provides a symmetrical view of confidentiality and integrity in the framework of *downgrading policies*. Although it is a common belief that confidentiality and integrity are duals, there are many aspects where integrity policies are fundamentally different from confidentiality policies. Integrity policies should precisely describe the computations on data in addition to the sources of data. The traditional noninterference-based approach provides no integrity guarantees for untrusted code, and is often too strong when dealing with implicit information flow.

This paper extended the framework of *downgrading policies* by presenting an more expressive model of integrity policies, where each label describe a set of possible functions that could have computed the data in the past. The presentations of confidentiality policies and integrity policies are mostly symmetrical. Traditional security levels for information-flow integrity such as tainted and untainted can be elegantly interpreted in this framework.

The asymmetry between confidentiality and integrity is shown in the ordering of security labels and also in the formalization of security goals. The interpretation and the ordering of integrity labels show the reason that untainted provides a weak security guarantee and suggests the use of more precise integrity labels instead of untainted. The definition of the security goal for integrity permits information leak through control flow yet provides formal, intuitive and practically useful security guarantees.

References

[1] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[2] 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.

[3] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, September 2003.

[4] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 158–170, January 2005.

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

[6] 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.

[7] 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.

[8] Charles P. Pfleeger. *Security in Computing*, pages 5–6. Prentice-Hall, 1997. Second Edition.

[9] 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.

[10] 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.

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

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

[13] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.