A Type System for Data-Flow Integrity on Windows Vista

Avik Chaudhuri
University of California at Santa Cruz
avik@cs.ucsc.edu

Prasad Naldurg Sriram Rajamani
Microsoft Research India
{prasadn,sriram}@microsoft.com

Abstract

The Windows Vista operating system implements an interesting model of multi-level integrity. We observe that in this model, trusted code must participate in any information-flow attack. Thus, it is possible to eliminate such attacks by statically restricting trusted code. We formalize this model by designing a type system that can efficiently enforce data-flow integrity on Windows Vista. Typechecking guarantees that objects whose contents are statically trusted never contain untrusted values, regardless of what untrusted code runs in the environment. Some of Windows Vista's runtime access checks are necessary for soundness; others are redundant and can be optimized away.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Access controls, Information flow controls, Verification; D.2.4 [*Software Engineering*]: Program Verification—Correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques, Invariants, Mechanical verification

General Terms Security, Verification, Languages, Theory

Keywords dynamic access control, data-flow integrity, hybrid type system, explicit substitution

1. Introduction

Commercial operating systems are seldom designed to prevent information-flow attacks. Not surprisingly, such attacks are the source of many serious security problems in these systems [45]. Microsoft's Windows Vista operating system implements an integrity model that can potentially prevent such attacks. In some ways, this model resembles other, classical models of multi-level integrity [9]—every process and object[1] is tagged with an integrity label, the labels are ordered by levels of trust, and access control is enforced across trust boundaries. In other ways, it is radically different. While Windows Vista's access control prevents low-integrity processes from writing to high-integrity objects, it does not prevent high-integrity processes from reading low-integrity objects. Further, Windows Vista's integrity labels are dynamic—labels of processes and objects can change at runtime. This model allows processes at different trust levels to communicate, and allows dynamic access control. At the same time, it admits various information-flow attacks. Fortunately, it turns out that such attacks require the participation of trusted processes, and can be eliminated by code analysis.

In this paper, we provide a formalization of Windows Vista's integrity model. In particular, we specify an information-flow property called *data-flow integrity* (DFI), and present a static type system that can enforce DFI on Windows Vista. Roughly, DFI prevents any flow of data from the environment to objects whose contents are trusted. Our type system relies on Windows Vista's runtime access checks for soundness. The key idea in the type system is to maintain a static lower-bound label S for each object. While the dynamic label of an object can change at runtime, the type system ensures that it never goes below S, and the object never contains a value that flows from a label lower than S. The label S is declared by the programmer. Typechecking requires no other annotations, and can be mechanized by an efficient algorithm.

By design, DFI does not prevent implicit flows [19]. Thus DFI is weaker than noninterference [24]. Unfortunately, it is difficult to enforce noninterference on a commercial operating system such as Windows Vista. Implicit flows abound in such systems. Such flows arise out of frequent, necessary interactions between trusted code and the environment. They also arise out of covert control channels which, given the scope of such systems, are impossible to model sufficiently. Instead, DFI focuses on explicit flows [19]. This focus buys a reasonable compromise—DFI prevents a definite class of attacks, and can be enforced efficiently on Windows Vista. Several successful tools for malware detection follow this approach [12, 54, 48, 50, 17, 38], and a similar approach guides the design of some recent operating systems [20, 59].

Our definition of DFI is dual to standard definitions of secrecy based on explicit flows—while secrecy prevents sensitive values from flowing to the environment, DFI prevents the flow of values from the environment to sensitive objects. Since there is a rich literature on type-based and logic-based analysis for such definitions of secrecy [11, 3, 49, 13], it makes sense to adapt this analysis for DFI. Such an adaptation works, but requires some care. Unlike secrecy, DFI cannot be enforced without runtime checks. In particular, access checks play a crucial role by restricting untrusted processes that may run in the environment. Further, while secrecy prevents any flow of high-security information to the environment, DFI allows certain flows of low-security information from the environment. We need to introduce new technical devices for this purpose, including a technique based on *explicit substitution* [4] to track precise sources of values. This device is required not only to specify DFI precisely but also to prove that our type system enforces DFI.

We design a simple higher-order process calculus that simulates Windows Vista's security environment [32, 18, 44]. (The design of this language is discussed in detail in Section 6.) In this language, processes can fork new processes, create new objects, change the labels of processes and objects, and read, write, and execute ob-

[1] In this context, an object may be a file, a channel, a memory location, or indeed any reference to data or executable code.

PLAS'08, June 8, 2008, Tucson, Arizona, USA.

jects in exactly the same ways as Windows Vista allows. Our type system exploits Windows Vista's runtime access checks to enforce DFI, and can recognize many correct programs. At the same time, our type system subsumes Windows Vista's execution controls, allowing them to be optimized away.

1.1 Summary of contributions

To sum up, we make the following main contributions in this paper:

- We propose DFI as a practical multi-level integrity property in the setting of Windows Vista, and formalize DFI using a semantic technique based on explicit substitution.
- We present a type system that can efficiently enforce DFI on Windows Vista. Typechecking guarantees DFI regardless of what untrusted code runs in the environment.
- We show that while most of Windows Vista's runtime access checks are required to enforce DFI, Windows Vista's execution controls are not necessary and can be optimized away.

1.2 Outline

The rest of this paper is organized as follows. In Section 2, we introduce Windows Vista's security environment, and show how DFI may be violated in that environment. In Section 3, we design a calculus that simulates Windows Vista's security environment, equip the calculus with a semantics based on explicit substitution, and formalize DFI in the calculus. In Section 4, we present a system of integrity types and effects for this calculus. In Section 5, we prove soundness and other properties of typing. Finally, in Section 6, we discuss limitations and contributions with respect to related work and conclude. Supplementary material, including proof details and an efficient typechecking algorithm, appear in the full version of the paper [15] available online at `http://arxiv.org/abs/0803.3230`.

2. Windows Vista's integrity model

In this section, we provide a brief overview of Windows Vista's integrity model.[2] In particular, we introduce Windows Vista's security environment [32, 18, 44], and show how DFI may be violated in that environment. We observe that such violations require the participation of trusted processes. Intuitively, the responsibility of security lies with trusted users. Our type system provides a way for such users to manage this responsibility automatically.

2.1 Windows Vista's security environment

In Windows Vista, every process and object is tagged with a dynamic integrity label. We indicate such labels in brackets ($\_$) below. Labels are related by a total order $\sqsubseteq$, meaning "at most as trusted as". Let a range over processes, ω over objects, and P, O over labels. Processes can fork new processes, create new objects, change the labels of processes and objects, and read, write, and execute objects. In particular, a process with label P can:

(i) fork a new process $a(\mathsf{P})$;

(ii) create a new object $\omega(\mathsf{P})$;

(iii) lower its own label;

(iv) change the label of an object $\omega(\mathsf{O})$ to O' iff $\mathsf{O} \sqcup \mathsf{O}' \sqsubseteq \mathsf{P}$;

(v) read an object $\omega(\mathsf{O})$;

(vi) write an object $\omega(\mathsf{O})$ iff $\mathsf{O} \sqsubseteq \mathsf{P}$;

(vii) execute an object $\omega(\mathsf{O})$ by lowering its own label to $\mathsf{P} \sqcap \mathsf{O}$.

[2] Windows Vista further implements a discretionary access control model, which we ignore in this paper.

Rules (i) and (ii) are straightforward. Rule (iii) is guided by the principle of least privilege [35], and is used in Windows Vista to implement a feature called *user access control* (UAC) [44, 53]. This feature lets users execute commands with lower privileges when appropriate. For example, when a system administrator opens a new shell (typically with label High), a new process is forked with label Medium; the shell is then run by the new process. When an Internet browser is opened, it is always run by a new process whose label is lowered to Low; thus any code that gets run by the browser gets the label Low—by Rule (i)—and any file that is downloaded by the browser gets the label Low—by Rule (ii).

Rules (iv) and (v) facilitate dynamic access control and communication across trust boundaries, but can be dangerous if not used carefully. (We show some attacks to illustrate this point below.) In particular, Rule (iv) allows trusted processes to protect unprotected objects by raising their labels. (Users are required to confirm such protections via the user interface.) Rule (v) allows processes to read objects at lower trust levels.

Rule (vi) protects objects from being written by processes at lower trust levels. Thus, for example, untrusted code forked by a browser cannot touch local user files. User code cannot modify registry keys protected by a system administrator. Rule (vii) is part of UAC; it prevents users from accidentally launching less trusted executables with higher privileges. For example, a virus downloaded from the Internet cannot run in a trusted user shell. Neither can system code dynamically link user libraries.

2.2 Some attacks

We now show some attacks that remain possible in this environment. Basically, these attacks exploit Rules (iv) and (v) to bypass Rules (vi) and (vii).

(**Write and copy**) By Rule (vi), $a(\mathsf{P})$ cannot modify $\omega(\mathsf{O})$ if $\mathsf{P} \sqsubset \mathsf{O}$. However, $a(\mathsf{P})$ can modify some object $\omega'(\mathsf{P})$, and then some process $b(\mathsf{O})$ can copy $\omega'(\mathsf{P})$'s content to $\omega(\mathsf{O})$. Thus, Rule (iv) can be exploited to bypass Rule (vi).

(**Copy and execute**) By Rule (vii), $a(\mathsf{P})$ cannot execute $\omega(\mathsf{O})$ at P if $\mathsf{O} \sqsubset \mathsf{P}$. However, $a(\mathsf{P})$ can copy $\omega(\mathsf{O})$'s content to some object $\omega'(\mathsf{P})$ and then execute $\omega'(\mathsf{P})$. Thus, Rule (iv) can be exploited to bypass Rule (vii).

(**Unprotect, write, and protect**) By Rule (vi), $a(\mathsf{P})$ cannot modify $\omega(\mathsf{O})$ if $\mathsf{P} \sqsubset \mathsf{O}$. However, some process $b(\mathsf{O})$ can unprotect $\omega(\mathsf{O})$ to $\omega(\mathsf{P})$, then $a(\mathsf{P})$ can modify $\omega(\mathsf{P})$, and then $b(\mathsf{O})$ can protect $\omega(\mathsf{P})$ back to $\omega(\mathsf{O})$. Thus, Rule (v) can be exploited to bypass Rule (vi).

(**Copy, protect, and execute**) By Rule (vii), $a(\mathsf{P})$ cannot execute $\omega(\mathsf{O})$ at P if $\mathsf{O} \sqsubset \mathsf{P}$. However, some process $b(\mathsf{O})$ can copy $\omega(\mathsf{O})$'s content to an object $\omega'(\mathsf{O})$, and then $a(\mathsf{P})$ can protect $\omega'(\mathsf{O})$ to $\omega'(\mathsf{P})$ and execute $\omega'(\mathsf{P})$. Thus, Rules (iv) and (v) can be exploited to bypass Rule (vii).

All of these attacks can violate DFI; however, we observe that access control forces the participation of a trusted process (one with the higher label) in any such attack.

- In (**Write and copy**) or (**Unprotect, write, and protect**), suppose that the contents of $\omega(\mathsf{O})$ are trusted, and P is the label of untrusted code, with $\mathsf{P} \sqsubset \mathsf{O}$. Then data can flow from $a(\mathsf{P})$ to $\omega(\mathsf{O})$, violating DFI, as above. Fortunately, some process $b(\mathsf{O})$ can be blamed here.
- In (**Copy and execute**) or (**Copy, protect, and execute**), suppose that the contents of some object $\omega''(\mathsf{P})$ are trusted, and O is the label of untrusted code, with $\mathsf{O} \sqsubset \mathsf{P}$. Then data can flow from some process $b(\mathsf{O})$ to $\omega''(\mathsf{P})$, violating DFI, as follows: $b(\mathsf{O})$ packs code to modify $\omega''(\mathsf{P})$ and writes the code to

$\omega(\mathsf{O})$, and $a(\mathsf{P})$ unpacks and executes that code at P, as above. Fortunately, $a(\mathsf{P})$ can be blamed here.

Our type system can eliminate such attacks by restricting trusted processes (Section 4). (The type system does not restrict untrusted code running in the environment.) Conceptually, this guarantee can be cast as Wadler and Findler's "*well-typed programs can't be blamed*" [52]. We rely on the fact that a trusted process can be blamed for any violation of DFI; it follows that if all trusted processes are well-typed, there cannot be any violation of DFI.

3. A calculus for analyzing DFI on Windows Vista

To formalize our approach, we now design a simple higher-order process calculus that simulates Windows Vista's security environment. We introduce the syntax and informal semantics, and present some examples of programs and attacks in the language. We then present a formal semantics, guided by a precise characterization of explicit flows.

3.1 Syntax and informal semantics

Several simplifications appear in the syntax of the language. We describe processes by their code. We use variables as object names, and let objects contain packed code or names of other objects. We enforce a mild syntactic restriction on nested packing, which makes typechecking significantly more efficient [15] (see below). Finally, we elide conditionals—for our purposes, the code

$$\mathsf{if}\ \texttt{condition}\ \mathsf{then}\ a\ \mathsf{else}\ b$$

can be conservatively analyzed by composing a and b in parallel. (DFI is a *safety property* in the sense of [7], and the safety of the latter code implies that of the former. We discuss this point in more detail in Section 3.3.)

Values include variables, unit, and packed expressions. Expressions include those for forking new processes, creating new objects, changing the labels of processes and objects, and reading, writing, and executing objects. They also include standard expressions for evaluation and returning results (see Gordon and Hankin's concurrent object calculus [25]).

$f, g ::=$	expression
$f \Rsh g$	fork
t	action
$\mathsf{let}\ x = f\ \mathsf{in}\ g$	evaluation
r	result
$t ::=$	action
$\mathsf{new}(x\ \#\ \mathsf{S})$	create object
$[\mathsf{P}]\ a$	change process label
$\langle \mathsf{O} \rangle\ \omega$	change object label
$!\omega$	read object
$\omega := x$	write object
$\mathsf{exec}\ \omega$	execute object
$r ::=$	result
$x, y, z, \ldots, \omega$	variable
unit	unit

$a, b ::=$	process
$a \Rsh b$	fork
t	action
$\mathsf{let}\ x = a\ \mathsf{in}\ b$	evaluation
u	value
$u, v ::=$	value
r	result
$\mathsf{pack}(f)$	packed expression

Syntactically, we distinguish between processes and expressions: while every expression is a process, not every process is an expression. For example, $\mathsf{pack}(f)$ is not an expression, while $[\mathsf{P}]\ \mathsf{pack}(f)$ is. Only expressions can be packed. In particular, a process cannot be of the form $\mathsf{pack}(\mathsf{pack}(\ldots))$. This distinction does not reduce expressivity, since such a process can be expressed in the language as $\mathsf{let}\ x = \mathsf{pack}(\ldots)\ \mathsf{in}\ \mathsf{pack}(x)$. The benefits of this distinction become clear in Section 5, where we discuss an algorithm for typechecking. However, for the bulk of the paper, the reader may overlook this distinction—neither the semantics nor the type system depend on it.

Processes have the following informal meanings.

- $a \Rsh b$ forks a new process a with the current process label and continues as b (see Rule (i)).
- $\mathsf{new}(x\ \#\ \mathsf{S})$ creates a new object ω with the current process label, initializes ω with x, and returns ω (see Rule (ii)); the annotation S is used by the type system (Section 4) and has no runtime significance.
- $[\mathsf{P}]\ a$ changes the current process label to P and continues as a; it blocks if the current process label is lower than P (see Rule (iii)).
- $\langle \mathsf{O} \rangle\ \omega$ changes ω's label to O and returns unit; it blocks if ω is not bound to an object at runtime, or the current process label is lower than ω's label or O (see Rule (iv)).
- $!\omega$ returns the value stored in ω; it blocks if ω is not bound to an object at runtime (see Rule (v)).
- $\omega := x$ writes the value x to ω and returns unit; it blocks if ω is not bound to an object at runtime, or if the current process label is lower than ω's label (see Rule (vi)).
- $\mathsf{exec}\ \omega$ unpacks the value stored in ω to a process f, lowers the current process label with ω's label, and executes f; it blocks if ω is not bound to an object at runtime or if the value stored in ω is not a packed expression (see Rule (vii)).
- $\mathsf{let}\ x = a\ \mathsf{in}\ b$ executes a, binds the value returned by a to x, and continues as b with x bound.
- u returns itself.

3.2 Programming examples

We now consider some programming examples in the language. We assume that Low, Medium, High, and $\top$ are labels, ordered in the obvious way. We assume that the top-level process always runs with $\top$, which is the most trusted label.

Example 3.1. Suppose that a Medium user opens an Internet browser `ie.exe` with Low privileges (recall UAC), and clicks on a `url` that contains `virus.exe`; the virus contains code to overwrite the command shell executable `cmd.exe`, which has label $\top$.

$$\begin{aligned}
p_1 \triangleq\ & \mathsf{let}\ \texttt{cmd.exe} = \mathsf{new}(\ldots\ \#\ \top)\ \mathsf{in} \\
& \mathsf{let}\ \texttt{url} = [\mathsf{Low}]\ \mathsf{new}(\ldots\ \#\ \mathsf{Low})\ \mathsf{in} \\
& \mathsf{let}\ \texttt{binIE} = \mathsf{pack}(\mathsf{let}\ x = !\texttt{url}\ \mathsf{in}\ \mathsf{exec}\ x)\ \mathsf{in} \\
& \mathsf{let}\ \texttt{ie.exe} = \mathsf{new}(\texttt{binIE}\ \#\ \top)\ \mathsf{in} \\
& \\
& [\mathsf{Medium}]\ (\ldots \Rsh [\mathsf{Low}]\ \mathsf{exec}\ \texttt{ie.exe}) \Rsh \\
& [\mathsf{Low}]\ (\mathsf{let}\ \texttt{binVirus} = \mathsf{pack}(\texttt{cmd.exe} := \ldots)\ \mathsf{in} \\
& \qquad \mathsf{let}\ \texttt{virus.exe} = \mathsf{new}(\texttt{binVirus}\ \#\ \mathsf{Low})\ \mathsf{in} \\
& \qquad \texttt{url} := \texttt{virus.exe} \Rsh \\
& \qquad \ldots)
\end{aligned}$$

This code may eventually reduce to

$$\begin{aligned}
q_1 \triangleq\ & [\mathsf{Medium}]\ (\ldots \Rsh [\mathsf{Low}]\ \texttt{cmd.exe} := \ldots) \Rsh \\
& [\mathsf{Low}]\ (\ldots)
\end{aligned}$$

However, at this point the write to cmd.exe blocks due to access control. (Recall that a process with label Low cannot write to an object with label $\top$.)

Example 3.2. Next, consider the following attack, based on the **(Copy, protect, and execute)** attack in Section 2.2. A Medium user downloads a virus from the Internet that contains code to erase the user's home directory (home), and saves it by default in setup.exe. A High administrator protects and executes setup.exe.

$$
\begin{array}{l}
p_2 \triangleq \ \mathsf{let}\ \mathtt{url} = [\mathsf{Low}]\ \mathsf{new}(\ldots \#\ \mathsf{Low})\ \mathsf{in} \\
\quad \mathsf{let}\ \mathtt{setup.exe} = [\mathsf{Low}]\ \mathsf{new}(\ldots \#\ \mathsf{Low})\ \mathsf{in} \\
\quad \mathsf{let}\ \mathtt{binIE} = \mathsf{pack}(\mathsf{let}\ z = \ !\mathtt{url}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad \mathsf{let}\ x = \ !z\ \mathsf{in}\ \mathtt{setup.exe} := \ x)\ \mathsf{in} \\
\quad \mathsf{let}\ \mathtt{ie.exe} = \mathsf{new}(\mathtt{binIE} \# \top)\ \mathsf{in} \\
\quad \mathsf{let}\ \mathtt{home} = [\mathsf{Medium}]\ \mathsf{new}(\ldots \#\ \mathsf{Medium})\ \mathsf{in} \\
\quad \mathsf{let}\ \mathtt{empty} = \mathsf{unit}\ \mathsf{in} \\
\\
\quad [\mathsf{High}]\ (\cdots \Rsh \\
\qquad \mathsf{let}\ \_ = \langle \mathsf{High} \rangle\ \mathtt{setup.exe}\ \mathsf{in} \\
\qquad \mathsf{exec}\ \mathtt{setup.exe})\ \Rsh \\
\quad [\mathsf{Medium}]\ (\ldots\ \Rsh\ [\mathsf{Low}]\ \mathsf{exec}\ \mathtt{ie.exe})\ \Rsh \\
\quad [\mathsf{Low}]\ (\mathsf{let}\ \mathtt{binVirus} = \mathsf{pack}(\mathtt{home} := \mathtt{empty})\ \mathsf{in} \\
\qquad \mathsf{let}\ \mathtt{virus.exe} = \mathsf{new}(\mathtt{binVirus} \# \mathsf{Low})\ \mathsf{in} \\
\qquad \mathtt{url} := \mathtt{virus.exe}\ \Rsh \\
\qquad \ldots)
\end{array}
$$

This code may eventually reduce to

$$
\begin{array}{rl}
q_2 \ \triangleq & [\mathsf{High}]\ (\ldots\ \Rsh\ \mathtt{home} := \mathtt{empty})\ \Rsh \\
& [\mathsf{Medium}]\ (\ldots)\ \Rsh \\
& [\mathsf{Low}]\ (\ldots)
\end{array}
$$

The user's home directory may be erased at this point. (Recall that access control does not prevent a process with label High from writing to an object with label Medium.)

Here the administrator is required to confirm the protection of setup.exe via the user interface. Our type system can detect that this protection is dangerous, and warn the administrator.

3.3 An overview of DFI

Informally, DFI requires that objects whose contents are trusted at some label S never contain values that flow from labels lower than S. In Example 3.1, we trust the contents of cmd.exe at label $\top$, as declared by the static annotation $\top$. DFI is *not* violated in this example, since access control prevents the flow of data from Low to cmd.exe. On the other hand, in Example 3.2, we trust the contents of home at label Medium. DFI *is* violated in this example, since the value empty flows from Low to home.

By design, DFI is a safety property [7]—roughly, it can be defined as a set of behaviors such that for any behavior that is not in that set, there is some finite prefix of that behavior that is not in that set. To that end, DFI considers only *explicit* flows of data. Denning and Denning characterize explicit flows [19] roughly as follows: a flow of x is explicit if and only if the flow depends abstractly on x (that is, it depends on the existence of x, but not on the value x). Thus, for example, the violation of DFI in Example 3.2 does not depend on the value empty—*any* other value causes the same violation. Conversely, empty is not dangerous in itself. Consider the reduced process q_2 in Example 3.2. Without any knowledge of execution history, we cannot conclude that DFI is violated in q_2. Indeed, it is perfectly legitimate for a High-process to execute the code home := empty intentionally, say as part of administration. However, in Example 3.2, we know that this code is executed by unpacking some code designed by a Low-process. The violation of DFI is *due to this history*.

It follows that in order to detect violations of DFI, we must distinguish between various instances of a value, and track the sources of those instances during execution. We maintain this execution history in the operational semantics (Section 3.4), by a technique based on explicit substitution [4].

Before we move on, let us ease the tension between DFI and conditionals. In general, conditionals can cause implicit flows [19]; a flow of x can depend on the value x if x appears in the condition of some code that causes that flow. For example, the code

$$\mathsf{if}\ x = \mathtt{zero}\ \mathsf{then}\ \omega := \mathtt{zero}\ \mathsf{else}\ \omega := \mathtt{one}$$

causes an implicit flow of x to ω that depends on the value x. We can abstract away this dependency by interpreting the code if condition then a else b as the parallel composition of a and b. Recall that DFI is a safety property. Following [34], the safety of this parallel composition can be expressed by the logical formula $F \triangleq F_a \wedge F_b$, where F_a is the formula that expresses the safety of a, and F_b is the formula that expresses the safety of b. Likewise, the safety of if condition then a else b can be expressed by the formula $F' \triangleq (\mathtt{condition} \Rightarrow F_a) \wedge (\neg \mathtt{condition} \Rightarrow F_b)$. Clearly, we have $F \Rightarrow F'$, so that the code if condition then a else b is a refinement of the parallel composition of a and b. It is well-known that safety is preserved under refinement [34].

But implicit flows are of serious concern in many applications; one may wonder whether focusing on explicit flows is even desirable. Consider the code above; the implicit flow from x to ω violates noninterference, if x is an untrusted value and the contents of ω are trusted. In contrast, DFI is *not* violated in the code

$$\omega := \mathtt{zero}\ \Rsh\ \omega := \mathtt{one}$$

if zero and one are trusted values. Clearly, DFI ignores the implicit flow from x to ω. But this may be fine—DFI can be used to prove an invariant such as "the contents of ω are either zero or one". Note that the code

$$\omega := x$$

does not maintain this invariant, since x may be an arbitrary value. Thankfully, DFI *is* violated in this code.

3.4 An operational semantics that tracks explicit flows

We now present a chemical-style operational semantics for the language, that tracks explicit flows.[3] We begin by extending the syntax with some auxiliary forms.

$a, b ::=$	process
$\cdots$	source process
$\omega \overset{\mathsf{O}}{\mapsto} x$	store
$(\nu x/\mu@\mathsf{P})\ a$	explicit substitution
$\mu ::=$	substituted value
u	value
$\mathsf{new}(x \# \mathsf{S})$	object initialization

The process $\omega \overset{\mathsf{O}}{\mapsto} x$ asserts that the object ω contains x and is protected with label O. A key feature of the semantics is that objects store values "by instance"—only variables may appear in stores. We use explicit substitution to track and distinguish between the sources of various instances of a substituted value. Specifically, the process $(\nu x/\mu@\mathsf{P})\ a$ creates a fresh variable x, records that x is bound to μ by a process with label P, and continues as a with x

[3] This presentation is particularly convenient for defining and proving DFI; a concrete implementation of the language may rely on a lighter semantics that does not track explicit flows.

Local reduction $a \xrightarrow{\mathsf{P};\sigma} b$

(**Reduct evaluate**)

$$\mathsf{let}\ x = u\ \mathsf{in}\ a \xrightarrow{\mathsf{P};\sigma} (\nu x/u@\mathsf{P})\ a$$

(**Reduct new**)

$$\mathsf{new}(x\,\#\,\mathsf{S}) \xrightarrow{\mathsf{P};\sigma} (\nu\omega/\mathsf{new}(x\,\#\,\mathsf{S})@\mathsf{P})\ (\omega \overset{\mathsf{P}}{\mapsto} x \Rsh \omega)$$

(**Reduct read**)

$$\frac{\omega \overset{\sigma}{=} \omega'}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ !\omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh x}$$

(**Reduct write**)

$$\frac{\omega \overset{\sigma}{=} \omega' \qquad \mathsf{O} \sqsubseteq \mathsf{P}}{\omega \overset{\mathsf{O}}{\mapsto} \_ \Rsh\ \omega' := x \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \mathsf{unit}}$$

(**Reduct execute**)

$$\frac{\omega \overset{\sigma}{=} \omega' \qquad \mathsf{pack}(f) \in \sigma(x) \qquad \mathsf{P}' = \mathsf{P} \sqcap \mathsf{O}}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \mathsf{exec}\ \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ [\mathsf{P}']\ f}$$

(**Reduct un/protect**)

$$\frac{\omega \overset{\sigma}{=} \omega' \qquad \mathsf{O} \sqcup \mathsf{O}' \sqsubseteq \mathsf{P}}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \langle \mathsf{O}' \rangle\ \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}'}{\mapsto} x \Rsh\ \mathsf{unit}}$$

Structural equivalence $a \equiv b$

(**Struct bind**)

$$\mathcal{E}_{\mathsf{P};\sigma}[\![a\{x/y\}]\!]_{\mathsf{P}',\sigma'} \equiv \mathcal{E}_{\mathsf{P};\sigma}[\![(\nu x/y@\mathsf{P}')\ a]\!]_{\mathsf{P}',\sigma'}$$

(**Struct substitution**)

$$\frac{x \notin \mathtt{fv}(\mathcal{E}_{\mathsf{P},\sigma}) \cup \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) \qquad \mathtt{fv}(\mu) \cap \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) = \varnothing}{\mathcal{E}_{\mathsf{P};\sigma}[\![(\nu x/\mu@\mathsf{P}'')\ a]\!]_{\mathsf{P}',\sigma'} \equiv (\nu x/\mu@\mathsf{P}'')\ \mathcal{E}_{\mathsf{P},\{x/\mu@\mathsf{P}''\}\cup\sigma}[\![a]\!]_{\mathsf{P}',\sigma'}}$$

(**Struct fork**)

$$\frac{\mathtt{fv}(a) \cap \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) = \varnothing}{\mathcal{E}_{\mathsf{P};\sigma}[\![a \Rsh b]\!]_{\mathsf{P},\sigma'} \equiv a \Rsh \mathcal{E}_{\mathsf{P};\sigma}[\![b]\!]_{\mathsf{P},\sigma'}}$$

(**Struct store**)

$$[\mathsf{P}]\ (\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ a) \equiv \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ [\mathsf{P}]\ a$$

(**Struct equiv**)

$\equiv$ is an equivalence

Global reduction $a \xrightarrow{\mathsf{P};\sigma} b$

(**Reduct context**)

$$\frac{a \xrightarrow{\mathsf{P}';\sigma'} b}{\mathcal{E}_{\mathsf{P};\sigma}[\![a]\!]_{\mathsf{P}';\sigma'} \xrightarrow{\mathsf{P};\sigma} \mathcal{E}_{\mathsf{P};\sigma}[\![b]\!]_{\mathsf{P}';\sigma'}}$$

(**Reduct congruence**)

$$\frac{a \equiv a' \qquad a' \xrightarrow{\mathsf{P};\sigma} b' \qquad b' \equiv b}{a \xrightarrow{\mathsf{P};\sigma} b}$$

bound. Here x is an *instance* of μ and P is the *source* of x. If μ is a value, then this process is behaviorally equivalent to a with x substituted by μ. For example, in Example 3.2 the source of the instance of `empty` in `binVirus` is Low; this fact is described by rewriting the process q_2 as

$$(\nu x/\mathtt{empty}@\mathsf{Low})\ [\mathsf{High}]\ (\cdots \Rsh\ \mathtt{home} := x) \Rsh \ldots$$

DFI prevents this particular instance (x) of `empty` from being written to `home`; but it allows other instances whose sources are at least as trusted as Medium. The rewriting follows a structural equivalence rule (**Struct bind**), explained later in the section.

While explicit substitution has been previously used in language implementations, we seem to be the first to adapt this device to track data flow in a concurrent language. In particular, we use explicit substitution both to specify DFI (in Definitions 3.3 and 3.4) and to verify it statically (in proofs of Theorems 5.4 and 5.7). We defer a more detailed discussion on this technique to Section 6.

We call sets of the form $\{x_1/\mu_1@\mathsf{P}_1, \ldots, x_k/\mu_k@\mathsf{P}_k\}$ *substitution environments*.

Definition 3.3 (Explicit flows)**.** *A variable x flows from a label P or lower in a substitution environment σ, written $x \overset{\sigma}{\blacktriangledown} \mathsf{P}$, if $x/\mu@\mathsf{P}' \in \sigma$ for some μ and P' such that either $\mathsf{P}' \sqsubseteq \mathsf{P}$, or μ is a variable and (inductively) $\mu \overset{\sigma}{\blacktriangledown} \mathsf{P}$.*

In other words, x flows from a label P or lower if x is an instance of a value substituted at P or lower. In Definition 3.4 below, we formalize DFI as a property of objects, as follows: *an object is protected from label L if it never contains instances that flow from L or lower.* We define $\sigma(x)$ to be the set of values in σ that x is an instance of: $x \in \sigma(x)$, and if (inductively) $y \in \sigma(x)$ and $y/u@\_ \in \sigma$ for some y and u, then $u \in \sigma(x)$. The operational semantics ensures that substitution environments accurately associate instances of values with their runtime sources.

We now present rules for local reduction, structural equivalence, and global reduction. Reductions are of the form $a \xrightarrow{\mathsf{P};\sigma} b$, meaning that "process a may reduce to process b with label P in substitution environment σ". Structural equivalences are of the form $a \equiv b$, meaning that "process a may be rewritten as process b". The notions of free and bound variables (`fv` and `bv`) are standard. We write $x \overset{\sigma}{=} y$ if $\sigma(x) \cap \sigma(y) \neq \varnothing$, that is, there is a value that both x and y are instances of.

We first look at the local reduction rules. In (**Reduct evaluate**), a substitution binds x to the intermediate value u and associates x with its runtime source P. (**Reduct new**) creates a new store denoted by a fresh variable ω, initializes the store, and returns ω; a substitution binds ω to the initialization of the new object and associates ω with its runtime source P. The value x and the trust annotation S in the initialization are used by the type system (Section 4). The remaining local reduction rules describe reactions with a store, following the informal semantics.

Next, we define evaluation contexts [21]. An evaluation context is of the form $\mathcal{E}_{\mathsf{P};\sigma}$, and contains a hole of the form $\bullet_{\mathsf{P}';\sigma'}$; the context yields a process that executes with label P in substitution environment σ, if the hole is plugged by a process that executes with label P' in substitution environment σ'.

$\mathcal{E}_{\mathsf{P};\sigma} ::=$	evaluation context
$\bullet_{\mathsf{P};\sigma}$	hole
$\mathsf{let}\ x = \mathcal{E}_{\mathsf{P};\sigma}\ \mathsf{in}\ b$	sequential evaluation
$\mathcal{E}_{\mathsf{P};\sigma} \Rsh b$	fork left
$a \Rsh \mathcal{E}_{\mathsf{P};\sigma}$	fork right
$(\nu x/\mu@\mathsf{P}')\ \mathcal{E}_{\mathsf{P};\{x/\mu@\mathsf{P}'\}\cup\sigma}$	explicit substitution
$[\mathsf{P}']\ \mathcal{E}_{\mathsf{P}';\sigma} \quad (\mathsf{P}' \sqsubseteq \mathsf{P})$	lowering of process label

Evaluation can proceed sequentially inside let processes, and in parallel under forks [25]; it can also proceed under explicit substitutions and lowering of process labels. In particular, note how evaluation contexts build substitution environments from explicit substitutions, and labels from changes of process labels. We denote by $\mathcal{E}_{\mathsf{P};\sigma}[\![a]\!]_{\mathsf{P}';\sigma'}$ the process obtained by plugging the hole $\bullet_{\mathsf{P}';\sigma'}$ in $\mathcal{E}_{\mathsf{P};\sigma}$ with a.

Next, we look at the structural equivalence and global reduction rules. In (**Struct bind**), $a\{x/y\}$ is the process obtained from a by the usual capture-avoiding substitution of x by y. The rule states that explicit substitution may *invert* usual substitution to create instances as required. In particular, variables that appear in packed code can be associated with the label of the process that packs that code, even though those variables may be bound later—by (**Reduct evaluate**)—when that code is eventually unpacked at some other label. For example, the instance of `empty` in `binVirus` may be correctly associated with Low (the label at which it is packed) instead of High (the label at which it is unpacked). In combination, the rules (**Reduct evaluate**) and (**Struct bind**) track precise sources of values by explicit substitution.

By (**Struct substitution**), substitutions can float across contexts under standard scoping restrictions. By (**Struct fork**), forked processes can float across contexts [25], but must remain under the same process label. By (**Struct store**), stores can be shared across further contexts.

Reduction is extended with contexts and structural equivalence in the natural way.

Finally, we formalize DFI in our language, as promised.

Definition 3.4 (DFI)**.** *The object ω is protected from label L by process a if there is no process b, substitution environment σ, and instance x such that $a \Rsh [\mathsf{L}]\ b \xrightarrow{\top,\varnothing}{}^{\star} \mathcal{E}_{\top,\varnothing}[\![\omega \mapsto x]\!]_{\top,\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$.*

4. A type system to enforce DFI

We now show a type system to enforce DFI in the language. (The formal protection guarantee for well-typed code appears in Section 5.) We begin by introducing types and typing judgments. We then present typing rules and informally explain their properties. Finally, we consider some examples of typechecking. An efficient algorithm for typechecking is outlined in [15].

4.1 Types and effects

The core grammar of types is shown below. Here effects are simply labels; these labels belong to the same ordering $\sqsubseteq$ as in the operational semantics.

$\tau ::=$	type
$\quad \mathbf{Obj}(T)$	object
$\quad \nabla_{\mathsf{P}}.\, \mathbf{Bin}(T)$	packed code
$\quad \mathbf{Unit}$	unit
$T ::=$	static approximation
$\quad \tau^{\mathsf{E}}$	type and effect

- The type $\mathbf{Obj}(\tau^{\mathsf{S}})$ is given to an object that contains values of type τ. Such contents may not flow from labels lower than S; in other words, S indicates the trust on the contents of this object. DFI follows from the soundness of object types.
- The type $\nabla_{\mathsf{P}}.\, \mathbf{Bin}(\tau^{\mathsf{E}})$ is given to packed code that can be run with label P. Values returned by the code must be of type τ and may not flow from labels lower than E. In fact, our type system admits a subtyping rule that allows such code to be run in a typesafe manner with any label that is at most P.
- The effect E is given to a value that does not flow from labels lower than E.

Core typing judgments $\Gamma \vdash_{\mathsf{P}} a : T$

(**Typ unit**)

$$\Gamma \vdash_{\mathsf{P}} \mathsf{unit} : \mathbf{Unit}^{\mathsf{P}}$$

(**Typ variable**)

$$\frac{x : \tau^{\mathsf{E}} \in \Gamma}{\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E} \sqcap \mathsf{P}}}$$

(**Typ fork**)

$$\frac{\Gamma \vdash_{\mathsf{P}} a : \_ \qquad \Gamma \vdash_{\mathsf{P}} b : T}{\Gamma \vdash_{\mathsf{P}} a \Rsh b : T}$$

(**Typ limit**)

$$\frac{\Gamma \vdash_{\mathsf{P}'} a : T}{\Gamma \vdash_{\mathsf{P}} [\mathsf{P}']\ a : T}$$

(**Typ evaluate**)

$$\frac{\Gamma \vdash_{\mathsf{P}} a : T' \qquad \Gamma, x : T' \vdash_{\mathsf{P}} b : T}{\Gamma \vdash_{\mathsf{P}} \mathsf{let}\ x = a\ \mathsf{in}\ b : T}$$

(**Typ substitute**)

$$\frac{\Gamma \vdash_{\mathsf{P}'} \mu : T' \qquad \Gamma, x : T' \vdash_{\mathsf{P}} a : T}{\Gamma \vdash_{\mathsf{P}} (\nu x/\mu@\mathsf{P}')\ a : T}$$

(**Typ store**)

$$\frac{\{\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\text{-}}, x : \tau^{\mathsf{E}}\} \subseteq \Gamma \qquad \mathsf{S} \sqsubseteq \mathsf{O} \sqcap \mathsf{E}}{\Gamma \vdash_{\mathsf{P}} \omega \overset{\mathsf{O}}{\mapsto} x : \_^{\mathsf{P}}}$$

(**Typ new**)

$$\frac{\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E}} \qquad \mathsf{S} \sqsubseteq \mathsf{E}}{\Gamma \vdash_{\mathsf{P}} \mathsf{new}(x\,\#\,\mathsf{S}) : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{P}}}$$

(**Typ pack**)

$$\frac{\Gamma \vdash_{\mathsf{P}'} f : T \qquad \Box f}{\Gamma \vdash_{\mathsf{P}} \mathsf{pack}(f) : \nabla_{\mathsf{P}'}.\, \mathbf{Bin}(T)^{\mathsf{P}}}$$

(**Typ un/protect**)

$$\frac{\Gamma \vdash_{\mathsf{P}} \omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \qquad \mathsf{S} \sqsubseteq \mathsf{O}}{\Gamma \vdash_{\mathsf{P}} \langle \mathsf{O} \rangle\ \omega : \mathbf{Unit}^{\mathsf{P}}}\ \boxed{*\mathsf{P} \Rightarrow *\mathsf{E}}$$

(**Typ write**)

$$\frac{\Gamma \vdash_{\mathsf{P}} \omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}} \qquad \Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E}'} \qquad \mathsf{S} \sqsubseteq \mathsf{E}'}{\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Unit}^{\mathsf{P}}}\ \boxed{*\mathsf{P} \Rightarrow *\mathsf{E}}$$

(**Typ read**)

$$\frac{\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}} \in \Gamma}{\Gamma \vdash_{\mathsf{P}}\ !\omega : \tau^{\mathsf{S} \sqcap \mathsf{P}}}\ \boxed{*(\mathsf{P} \sqcap \mathsf{S}) \Rightarrow *\mathsf{E}}$$

(**Typ execute**)

$$\frac{\omega : \mathbf{Obj}((\nabla_{\mathsf{P}'}.\, \mathbf{Bin}(\tau^{\mathsf{E}'}))^{\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubseteq \mathsf{P}' \sqcap \mathsf{S}}{\Gamma \vdash_{\mathsf{P}} \mathsf{exec}\ \omega : \tau^{\mathsf{E}' \sqcap \mathsf{P}}}\ \boxed{*\mathsf{P} \Rightarrow *\mathsf{E}}$$

When creating an object, the programmer declares the trust on the contents of that object. Roughly, an object returned by $\mathsf{new}(\_ \# \mathsf{S})$ gets a type $\mathbf{Obj}(\_^{\mathsf{S}})$. For example, in Examples 3.1 and 3.2, we declare the trust $\top$ on the contents of `cmd.exe` and the trust Medium on the contents of `home`.

A typing environment Γ contains typing hypotheses of the form $x : T$. We assume that any variable has at most one typing hypothesis in Γ, and define $\mathtt{dom}(\Gamma)$ as the set of variables that have typing hypotheses in Γ. A typing judgment is of the form $\Gamma \vdash_{\mathsf{P}} a : T$, where P is the label of the process a, T is the type and effect of values returned by a, and $\mathtt{fv}(a) \subseteq \mathtt{dom}(\Gamma)$.

4.2 Core typing rules

We now present typing rules that enforce the core static discipline required for our protection guarantee. Some of these rules have side conditions that involve a predicate $*$ on labels. These conditions, which are marked in shaded boxes, are ignored in our first reading of these rules. (The predicate $*$ is true everywhere in the absence of a special label $\bot$, introduced in Section 4.4.) One of the rules has a condition that involves a predicate $\Box$ on expressions; we introduce that predicate in the discussion below. The typing rules preserve several invariants.

(1) Code that runs with a label P cannot return values that have effects higher than P.

(2) The contents of an object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot have effects lower than S.

(3) The dynamic label that protects an object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot be lower than S.

(4) An object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot be created at a label lower than S.

(5) Packed code of type $\nabla_{\mathsf{P}}.\ \mathbf{Bin}(\_)$ must remain well-typed when unpacked at any label lower than P.

Invariant (1) follows from our interpretation of effects. To preserve this invariant in (**Typ variable**), for example, the effect of x at P is obtained by lowering x's effect in the typing environment with P.

In (**Typ store**), typechecking is independent of the process label, that is, a store is well-typed if and only if it is so at any process label; recall that by (**Struct store**) stores can float across contexts, and typing must be preserved by structural equivalence. Further, (**Typ store**) introduces Invariants (2) and (3). Invariant (2) follows from our interpretation of static trust annotations. To preserve this invariant we require Invariant (3), which ensures that access control prevents code running with labels less trusted than S from writing to objects whose contents are trusted at S.

By (**Typ new**), the effect E of the initial content of a new object cannot be lower than S. Recall that by (**Reduct new**), the new object is protected with the process label P; since $\mathsf{P} \sqsupseteq \mathsf{E}$ by Invariant (1), we have $\mathsf{P} \sqsupseteq \mathsf{S}$, so that both Invariants (2) and (3) are preserved. Conversely, if $\mathsf{P} \sqsubset \mathsf{S}$ then the process does not typecheck; Invariant (4) follows.

Let us now look carefully at the other rules relevant to Invariants (2) and (3); these rules—combined with access control—are the crux of enforcing DFI. (**Typ write**) preserves Invariant (2), restricting trusted code from writing values to ω that may flow from labels lower than S. (Such code may not be restricted by access control.) Conversely, access control prevents code with labels lower than S from writing to ω, since by Invariant (3), ω's label is at least as trusted as S. (**Typ un/protect**) preserves Invariant (3), allowing ω's label to be either raised or lowered without falling below S. In (**Typ read**), the effect of a value read from ω at P is approximated by S—the least trusted label from which ω's contents may flow—and further lowered with P to preserve Invariant (1).

In (**Typ pack**), packing code requires work akin to proof-carrying code [40]. Type safety for the code is proved and "carried" in its type $\nabla_{\mathsf{P}'}.\ \mathbf{Bin}(T)$, independently of the current process label. Specifically, it is proved that when the packed code is unpacked by a process with label P', the value of executing that code has type and effect T. In Section 5, we show that such a proof in fact allows the packed code to be unpacked by any process with label $\mathsf{P} \sqsubseteq \mathsf{P}'$, and the type and effect of the value of executing that code can be related to T (Invariant (5)). This invariant is key to decidable and efficient typechecking [15]. Of course, code may be packed to run only at specific process labels, by requiring the appropriate label changes.

Preserving Invariant (5) entails, in particular, preserving Invariant (4) at all labels $\mathsf{P} \sqsubseteq \mathsf{P}'$. Since a new expression that is not guarded by a change of the process label may be run with any label P, that expression must place the least possible trust on the contents of the object it creates. This condition is enforced by predicate $\Box$:

$$\begin{aligned} \Box\mathsf{new}(x \# \mathsf{S}) &\triangleq \forall \mathsf{P}.\ \mathsf{S} \sqsubseteq \mathsf{P} \\ \Box(f \Rsh g) &\triangleq \Box f \wedge \Box g \\ \Box(\mathsf{let}\ x = f\ \mathsf{in}\ g) &\triangleq \Box f \wedge \Box g \\ \Box(\dots) &\triangleq \mathtt{true} \end{aligned}$$

(**Typ execute**) relies on Invariant (5); further, it checks that the label at which the code is unpacked (P) is at most as trusted as the label at which the code may have been packed (approximated by S). This check prevents privilege escalation—code that would perhaps block if run with a lower label cannot be packed to run with a higher label. For example, recall that in Example 3.2, the code `binVirus` is packed at Low and then copied into `setup.exe`. While a High-process can legitimately execute `home := empty` (so that the code is typed and is not blocked by access control), it should not run that code by unpacking `binVirus` from `setup.exe`. The type system prevents this violation. Let `setup.exe` be of type $\mathbf{Obj}((\nabla\_.\ \mathbf{Bin}(\_))^{\mathsf{S}})$. Then (**Typ store**) requires that $\mathsf{S} \sqsubseteq \mathsf{Low}$, and (**Typ execute**) requires that $\mathsf{High} \sqsubseteq \mathsf{S}$ (contradiction).

Because we do not maintain an upper bound on the dynamic label of an executable, we cannot rely on the lowering of the process label in (**Reduct execute**) to prevent privilege escalation. (While it is possible to extend our type system to maintain such upper bounds, such an extension does not let us typecheck any more correct programs than we already do.) In Section 5, we show that the lowering of the process label can in fact be safely eliminated.

In (**Typ evaluate**), typing proceeds sequentially, propagating the type and effect of the intermediate process to the continuation. (**Typ substitution**) is similar, except that the substituted value is typed under the process label recorded in the substitution, rather than under the current process label. In (**Typ limit**), the continuation is typed under the changed process label. In (**Typ fork**), the forked process is typed under the current process label.

4.3 Typing rules for stuck code

While the rules above rely on access control for soundness, they do not *exploit* runtime protection provided by access control to typecheck more programs. For example, the reduced process q_1 in Example 3.1 cannot yet be typed, although we have checked that DFI is not violated in q_1. Below, we introduce *stuck typing* to identify processes that provably block by access control at runtime. Stuck typing allows us to soundly type more programs by composition. (The general principle that is followed here is that narrowing the set of possible execution paths improves the precision of the analysis.) This powerful technique of combining static typing and dynamic access control for runtime protection is quite close to hybrid typechecking [22]. We defer a more detailed discussion of this technique to Section 6.

Stuck typing judgments $\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}$

(**Typ escalate stuck**)

$$\frac{\mathsf{P} \sqsubset \mathsf{P}'}{\Gamma \vdash_{\mathsf{P}} [\mathsf{P}']\, a : \mathbf{Stuck}}$$

(**Typ write stuck**)

$$\frac{\omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubset \mathsf{S}}{\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Stuck}} \ *\mathsf{E}$$

(**Typ un/protect stuck**)

$$\frac{\omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubset \mathsf{S} \sqcup \mathsf{O}}{\Gamma \vdash_{\mathsf{P}} \langle \mathsf{O} \rangle\, \omega : \mathbf{Stuck}} \ *\mathsf{E}$$

(**Typ subsumption stuck-I**)

$$\frac{\_ : \mathbf{Stuck} \in \Gamma}{\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}}$$

(**Typ subsumption stuck-II**)

$$\frac{\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}}{\Gamma \vdash_{\mathsf{P}} a : T}$$

We introduce the static approximation $\mathbf{Stuck}$ for processes that do not return values, but may have side effects.

$T ::=$	static approximation
$\cdots$	code
$\mathbf{Stuck}$	stuck process

We now present rules for stuck-typing. As before, in our first reading of these rules we ignore the side conditions in shaded boxes (which involve the predicate $*$). (**Typ write stuck**) identifies code that tries to write to an object whose static trust annotation S is higher than the current process label P. By Invariant (3), the label O that protects the object must be at least as high as S; thus $\mathsf{P} \sqsubset \mathsf{O}$ and the code must block at runtime due to access control. For example, let `cmd.exe` be of type $\mathbf{Obj}(\_^{\top})$ in Example 3.1. By (**Typ write stuck**), the code q_1 is well-typed since $\mathsf{Low} \sqsubset \top$. (**Typ un/protect stuck**) is similar to (**Typ write stuck**); it further identifies code that tries to raise the label of an object beyond the current process label. (**Typ escalate stuck**) identifies code that tries to raise the current process label. All such processes block at runtime due to access control.

By (**Typ subsumption stuck-I**), processes that are typed under stuck hypotheses are considered stuck as well. For example, this rule combines with (**Typ evaluate**) to trivially type a continuation b if the intermediate process a is identified as stuck. Finally, by (**Typ subsumption stuck-II**), stuck processes can have any type and effect, since they cannot return values.

4.4 Typing rules for untrusted code

Typing must guarantee protection in arbitrary environments. Since the protection guarantee is derived via a type preservation theorem, arbitrary untrusted code needs to be accommodated by the type system. We assume that untrusted code runs with a special label $\bot$, introduced into the total order by assuming $\bot \sqsubseteq \mathsf{L}$ for all L. We now present rules that allow arbitrary interpretation of types at $\bot$. By (**Typ subsumption $\bot$-I**), placing the static trust $\bot$ on the contents of an object amounts to assuming any type for those contents as required. By (**Typ subsumption $\bot$-II**), a value that has effect $\bot$ may be assumed to have any type as required. These rules provide the necessary flexibility for typing any untrusted code using the other typing rules. On the other hand, arbitrary subtyping with objects can in general be unsound—we now need to be careful when typing trusted code. For example, consider the code

Typing rules for untrusted code

(**Typ subsumption $\bot$-I**)

$$\frac{\Gamma, \omega : \mathbf{Obj}(\_^{\bot})^{\mathsf{E}} \vdash_{\mathsf{P}} a : T}{\Gamma, \omega : \mathbf{Obj}(\tau^{\bot})^{\mathsf{E}} \vdash_{\mathsf{P}} a : T}$$

(**Typ subsumption $\bot$-II**)

$$\frac{\Gamma, x : \_^{\bot} \vdash_{\mathsf{P}} a : T}{\Gamma, x : \tau^{\bot} \vdash_{\mathsf{P}} a : T}$$

$$\omega_2 \overset{\mathsf{High}}{\mapsto} x \ \Rsh\ \omega_1 \overset{\mathsf{Low}}{\mapsto} \omega_2 \ \Rsh\ [\mathsf{High}]\ \mathsf{let}\ z = !\omega_1\ \mathsf{in}\ z := u$$

A High-process reads the name of an object (ω_2) from a Low-object (ω_1), and then writes u to that object (ω_2). DFI is violated if ω_2 has type $\mathbf{Obj}(\_^{\mathsf{High}})$ and u flows from Low. Unfortunately, it turns out that this code can be typed under process label $\top$ and typing hypotheses

$$\omega_2 : \mathbf{Obj}(\tau_2^{\mathsf{High}})^{\top},\ \omega_1 : \mathbf{Obj}(\mathbf{Obj}(\tau_2^{\mathsf{High}})^{\bot})^{\top},\ x : \tau_2^{\mathsf{High}},\ u : \tau_1^{\mathsf{Low}}$$

Specifically, the intermediate judgment

$$z : \mathbf{Obj}(\tau_2^{\mathsf{High}})^{\bot}, \ldots, u : \tau_1^{\mathsf{Low}} \ \vdash_{\mathsf{High}}\ z := u : \_$$

can be derived by adjusting the type of z in the typing environment to $\mathbf{Obj}(\tau_1^{\mathsf{Low}})$ with (**Typ subsumption $\bot$-II**).

This source of unsoundness is eliminated if some of the effects in our typing rules are required to be trusted, that is, to be higher than $\bot$. Accordingly we introduce the predicate $*$, such that for any label L, $*\mathsf{L}$ simply means $\mathsf{L} \sqsupset \bot$. We now revisit the typing rules earlier in the section and focus on the side conditions in shaded boxes (which involve $*$). In some of those conditions, we care about trusted effects only if the process label is itself trusted. With these conditions, (**Typ write**) prevents typechecking the offending write above, since the effect of z in the typing environment is untrusted.

4.5 Compromise

The label $\bot$ introduced above is an artificial construct to tolerate a degree of "anarchy" in the type system. We may want to specify that a certain label (such as Low) acts like $\bot$, *i.e.*, is *compromised*. The typing judgment $\Gamma \vdash_{\mathsf{P}} a : T$ despite C allows us to type arbitrary code a running at a compromised label C by assuming that C is the same as $\bot$, *i.e.*, by extending the total order with $\mathsf{C} \sqsubseteq \bot$ (so that all labels that are at most as trusted as C collapse to $\bot$). We do not consider labels compromised at runtime (as in Gordon and Jeffrey's type system for conditional secrecy [27]); however we do not anticipate any technical difficulty in including runtime compromise in our type system.

4.6 Typechecking examples

We now show some examples of typechecking.

We begin with the program p_2 in Example 3.2. Recall that DFI is violated in p_2. Suppose that we try to derive the typing judgment

$$\cdots \vdash_{\top} p_2 : \_ \ \text{despite Low}$$

This amounts to deriving $\cdots \vdash_{\top} p_2 : \_$ by assuming $\mathsf{Low} \sqsubseteq \bot$.

As a first step, we apply (**Typ new**), (**Typ read**), (**Typ write**), (**Typ pack**), and (**Typ evaluate**), directed by syntax, until we have the following typing environment.

$$\begin{aligned}\Gamma \;=\; & \ldots, \\ & \texttt{url} : \mathbf{Obj}(\_^{\mathsf{Low}})^{\top}, \\ & \texttt{setup.exe} : \mathbf{Obj}(\_^{\mathsf{Low}})^{\top}, \\ & \texttt{binIE} : (\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^{\top}, \\ & \texttt{ie.exe} : \mathbf{Obj}((\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^{\top})^{\top}, \\ & \texttt{home} : \mathbf{Obj}(\_^{\mathsf{Medium}})^{\top} \\ & \texttt{empty} : \mathbf{Unit}^{\top}\end{aligned}$$

The only complication that may arise is in this step is in deriving an intermediate judgment

$$\ldots,\ z : \_^{\mathsf{Low}} \ \vdash_{\top}\ !z : \_$$

Here, we can apply (**Typ subsumption $\bot$-II**) to adjust the typing hypothesis of z to $\mathbf{Obj}(\_)^{\bot}$, so that (**Typ read**) may apply.

After this step, we need to derive a judgment of the form:

$$\Gamma \vdash_{\top} [\mathsf{High}]\ (\ldots)\ \Rsh\ [\mathsf{Medium}]\ (\ldots)\ \Rsh\ [\mathsf{Low}]\ (\ldots)$$

Now, we apply (**Typ fork**). We first check that the code $[\mathsf{Low}]\ (\ldots)$ is well-typed. (In fact, untrusted code is always well-typed, as we show in Section 5.) The judgment

$$\Gamma \vdash_{\mathsf{Low}} \texttt{home} := \texttt{empty} : \mathbf{Unit}$$

typechecks by (**Typ write stuck**). Thus, by (**Typ pack**) and (**Typ evaluate**), we add the following hypothesis to the typing environment.

$$\texttt{binVirus} : (\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^{\mathsf{Low}}$$

Let $T_{\texttt{binVirus}} = (\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^{\mathsf{Low}}$. Next, by (**Typ new**) and (**Typ evaluate**), we add the following hypothesis to the typing environment.

$$\texttt{virus.exe} : \mathbf{Obj}(T_{\texttt{binVirus}})^{\mathsf{Low}}$$

Finally, the judgment

$$\Gamma, \ldots, \texttt{virus.exe} : \mathbf{Obj}(T_{\texttt{binVirus}})^{\mathsf{Low}} \vdash_{\mathsf{Low}} \texttt{url} := \texttt{virus.exe}$$

can be derived by (**Typ write**), after massaging the typing hypothesis for `virus.exe` to the required $\_^{\mathsf{Low}}$ by (**Typ subsumption $\bot$-II**).

On the other hand, the process $[\mathsf{High}]\ (\ldots)$ does not typecheck; as seen above, an intermediate judgment

$$\Gamma \vdash_{\mathsf{High}} \mathsf{exec}\ \texttt{setup.exe} : \_$$

cannot be derived, since (**Typ execute**) does not apply.

To understand this situation further, let us consider some variations where (**Typ execute**) does apply. Suppose that the code exec z is forked in a new process whose label is lowered to Low. Then p_2 typechecks. In particular, the following judgment can be derived by applying (**Typ execute**).

$$\Gamma \vdash_{\mathsf{High}} [\mathsf{Low}]\ \mathsf{exec}\ \texttt{setup.exe} : \_$$

Fortunately, the erasure of `home` now blocks by access control at runtime, so DFI is not violated.

Next, suppose that the static annotation for `setup.exe` is High instead of Low, and `setup.exe` is initialized by a process with label High instead of Low. Then p_2 typechecks. In particular, the type of `setup.exe` in Γ becomes $\mathbf{Obj}(\_^{\mathsf{High}})$. We need to derive an intermediate judgment

$$\Gamma, \ldots, x : \_ \vdash_{\mathsf{Low}} \texttt{setup.exe} := x : \mathbf{Unit}$$

This judgment can be derived by applying (**Typ write stuck**) instead of (**Typ write**). Fortunately, the overwrite of `setup.exe` now blocks by access control at runtime, so DFI is not violated.

Finally, we sketch how typechecking fails for the violations of DFI described in Section 2.2.

(**Write and copy**) Let the type of ω be $\mathbf{Obj}(\_^{\mathsf{S}})$, where $\mathsf{O} \sqsupseteq \mathsf{S} \sqsupset \mathsf{P}$. Then the write to $\omega(\mathsf{O})$ does not typecheck, since the value to be written is read from $\omega'(\mathsf{P})$ and thus has some effect E such that $\mathsf{E} \sqsubseteq \mathsf{P}$, so that $\mathsf{E} \sqsubset \mathsf{S}$.

(**Copy and execute**) Let the type of ω' be $\mathbf{Obj}(\_^{\mathsf{S}'})$. If $\mathsf{S}' \sqsubseteq \mathsf{O}$ then the execution of $\omega'(\mathsf{P})$ by $a(\mathsf{P})$ does not typecheck, since $\mathsf{S}' \sqsubset \mathsf{P}$. If $\mathsf{S}' \sqsupset \mathsf{O}$ then the write to $\omega'(\mathsf{P})$ does not typecheck, since the value to be written is read from $\omega(\mathsf{O})$ and thus has some effect E such that $\mathsf{E} \sqsubseteq \mathsf{O}$, so that $\mathsf{E} \sqsubset \mathsf{S}'$.

(**Unprotect, write, and protect**) Let the type of ω be $\mathbf{Obj}(\_^{\mathsf{S}})$, where $\mathsf{O} \sqsupseteq \mathsf{S} \sqsupset \mathsf{P}$. Then the unprotection of $\omega(\mathsf{O})$ does not typecheck, since $\mathsf{P} \sqsubset \mathsf{S}$.

(**Copy, protect, and execute**) Let the type of ω' be $\mathbf{Obj}(\_^{\mathsf{S}'})$, where $\mathsf{S}' \sqsubseteq \mathsf{O}$. Then the execution of $\omega'(\mathsf{P})$ does not typecheck, since $\mathsf{S}' \sqsubset \mathsf{P}$.

5. Properties of typing

In this section we show several properties of typing, and prove that DFI is preserved by well-typed code under arbitrary untrusted environments. All proof details appear in [15].

We begin with the proposition that untrusted code can always be accommodated by the type system.

Definition 5.1 (Adversary)**.** *A* C*-adversary is any process of the form* $[\mathsf{C}]\ \_$ *that does not contain stores, explicit substitutions, and static trust annotations that are higher than* C.

Proposition 5.2 (Adversary completeness)**.** *Let* Γ *be any typing environment and* c *be any* C*-adversary such that* $\texttt{fv}(c) \subseteq \mathsf{dom}(\Gamma)$*. Then* $\Gamma \vdash_{\top} c : \_\ \mathsf{despite}\ \mathsf{C}$.

Proposition 5.2 provides a simple way to quantify over arbitrary environments. By (**Typ fork**) the composition of a well-typed process with any such environment remains well-typed, and thus enjoys all the properties of typing.

Next, we present a monotonicity property of typing that is key to decidable and efficient typechecking [15].

Proposition 5.3 (Monotonicity)**.** *The following inference rule is admissible.*

$$\frac{\Gamma \vdash_{\mathsf{P}'} f : \tau^{\mathsf{E}} \qquad \Box f \qquad \mathsf{P} \sqsubseteq \mathsf{P}'}{\Gamma \vdash_{\mathsf{P}} f : \tau^{\mathsf{E} \sqcap \mathsf{P}}}$$

This rule formalizes Invariant (5), and allows inference of "most general" types for packed code [15]. Further, it implies an intuitive proof principle—code that is proved safe to run with higher privileges remains safe to run with lower privileges, and conversely, code that is proved safe against a more powerful adversary remains safe against a less powerful adversary.

The key property of typing is that it is preserved by structural equivalence and reduction. Preservation depends delicately on the design of the typing rules, relying on the systematic maintenance of typing invariants. We write $\Gamma \vdash \sigma$, meaning that "the substitution environment σ is consistent with the typing environment Γ", if for all $x/\mu@\,\mathsf{P} \in \sigma$ there exists T such that $x : T \in \Gamma$ and $\Gamma \vdash_{\mathsf{P}} \mu : T$.

Theorem 5.4 (Preservation)**.** *Suppose that* $\Gamma \vdash \sigma$ *and* $\Gamma \vdash_{\mathsf{P}} a : \_$*. Then*

- *if* $a \equiv b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$*;*
- *if* $a \xrightarrow{\mathsf{P};\sigma} b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$

We now present our formal protection guarantee for well-typed code. We begin by strengthening the definition of DFI in Section 3.

In particular, we assume that part of the adversary is known and part of it is unknown. This assumption allows the analysis to exploit any sound typing information that may be obtained from the known part of the adversary. (As a special case, the adversary may be entirely unknown, of course. In this case, we recover Definition 3.4; see below.) Let Ω be the set of objects that require protection from labels L or lower. We let the unknown part of the adversary execute with some process label C ($\sqsubseteq \mathsf{L}$). We say that Ω is protected if no such adversary can write any instance that flows from L or lower, to any object in Ω.

Definition 5.5 (Strong DFI)**.** *A set of objects Ω is protected by code a from label L despite C ($\sqsubseteq \mathsf{L}$) if there is no $\omega \in \Omega$, C-adversary c, substitution environment σ, and instance x such that $a \Rsh c \xrightarrow{\top,\varnothing}{}^{\star} \mathcal{E}_{\top,\varnothing}[\![\omega \mapsto x]\!]_{\top,\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$.*

For example, we may want to prove that some code protects a set of High-objects from Medium despite (the compromised label) Low; then we need to show that no instance may flow from Medium or lower to any of those High-objects under any Low-adversary.

We pick objects that require protection based on their types and effects in the typing environment.

Definition 5.6 (Trusted objects)**.** *The set of objects whose contents are trusted beyond the label L in the typing environment Γ is $\{\omega \mid \omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma$ and $\mathsf{S} \sqcap \mathsf{E} \sqsupseteq \mathsf{L}\}$.*

Suppose that in some typing environment, Ω is the set of objects whose contents are trusted beyond label L, and C ($\sqsubseteq \mathsf{L}$) is compromised; we guarantee that Ω is protected by any well-typed code from L despite C.

Theorem 5.7 (Enforcement of strong DFI)**.** *Let Ω be the set of objects whose contents are trusted beyond L in Γ. Suppose that $\Gamma \vdash_{\top} a : \_$ despite C, where $\mathsf{C} \sqsubseteq \mathsf{L}$. Then a protects Ω from L despite C.*

In the special case where the adversary is entirely unknown, we simply consider L and C to be the same label.

The type system further enforces DFI for new objects, as can be verified by applying Theorem 5.4, (**Typ substitute**), and Theorem 5.7. Finally, the type system suggests a sound runtime optimization: whenever a well-typed process executes packed code in a trusted context, the current process label is already appropriately lowered for execution.

Theorem 5.8 (Redundancy of execution control)**.** *Suppose that $\Gamma \vdash_{\top} a : \_$ despite C and $a \xrightarrow{\top;\varnothing}{}^{\star} \mathcal{E}_{\top;\varnothing}[\![\omega \overset{\mathsf{O}}{\mapsto} \_ \Rsh \mathsf{exec}\ \omega']\!]_{\mathsf{P};\sigma}$ such that $\omega \overset{\sigma}{=} \omega'$ and $\mathsf{P} \sqsupseteq \mathsf{C}$. Then $\mathsf{P} \sqsubseteq \mathsf{O}$.*

It follows that the rule (**Reduct execute**) can be safely optimized as follows.

$$\frac{\omega \overset{\sigma}{=} \omega' \qquad \mathsf{pack}(f) \in \sigma(x)}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh \mathsf{exec}\ \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh f}$$

This optimization should not be surprising. Lowering the process label for execution aims to prevent trusted code from executing untrusted code in trusted contexts; our core static discipline on trusted code effectively subsumes this runtime control. On the other hand, write-access control cannot be eliminated by any discipline on trusted code, since that control is required to restrict untrusted code.

Lastly, typechecking can be efficiently mechanized thanks to Proposition 5.3 and our syntactic restriction on nested packing.

Theorem 5.9 (Typechecking)**.** *Given a typing environment Γ and code a with $\mathbb{L}$ distinct labels, the problem of whether there exists T such that $\Gamma \vdash_{\top} a : T$, is decidable in time $\mathcal{O}(\mathbb{L}|a|)$, where $|a|$ is the size of a.*

A typechecking algorithm is outlined in [15]. As usual, the algorithm builds constraints and then checks whether those constraints are satisfiable. The only complication is due to pack processes, which require "most general" types.

Briefly, the grammar of types is extended with type variables, and a distinguished label ? is introduced to denote an "unknown" label. Let a *typechecking environment* Δ be a typing environment augmented by simple type constraints, and a *label constraint* (a boolean formula with propositions of the form $\mathsf{L}_1 \sqsubseteq \mathsf{L}_2$). The following typechecking judgments are defined, with mutually recursive rules:

- $\Delta \vdash_{\mathsf{P}} a : T \rhd \Delta'$, where the label constraint in Δ' is true.
- $\Delta \vdash f : T \rhd \Delta'$, where Δ' contains a label constraint over ?.

The rules for $\Delta \vdash_{\mathsf{P}} a : T \rhd \Delta'$ build simple type constraints in Δ', following the original typing rules. To derive a judgment of the form $\Delta \vdash_{\mathsf{P}} \mathsf{pack}(f) : \_ \rhd \_$, we need to derive a judgment of the form $\Delta \vdash f : \_ \rhd \_$. The rules for $\Delta \vdash f : T \rhd \Delta'$ build label constraints from conditions on labels in the original typing rules; here, the implicit (unknown) process label is taken to be ?. To derive a judgment of the form $\Delta \vdash [\mathsf{P}]\ a : \_ \rhd \_$, we need to derive a judgment of the form $\Delta \vdash_{\mathsf{P}} a : \_ \rhd \_$. On the other hand, the syntactic restriction on expressions ensures that we do not need to consider judgments of the form $\Delta \vdash \mathsf{pack}(f) : \_ \rhd \_$.

Solving the simple type constraints built by a judgment of the form $\Delta \vdash_{\mathsf{P}} a : \_ \rhd \_$ takes time $\mathcal{O}(|a|)$; solving the label constraint built by a judgment of the form $\Delta \vdash f : \_ \rhd \_$ takes time $\mathcal{O}(\mathbb{L}|f|)$. The running time of the typechecking algorithm follows by a straightforward inductive argument.

6. Limitations, related work, and discussion

In this paper we formalize DFI—a multi-level integrity property based on explicit flows—and present a type system that can efficiently enforce DFI in a language that simulates Windows Vista's security environment.

Not surprisingly, our type system is only a conservative technique to enforce DFI—while every program that typechecks is guaranteed to satisfy DFI (as stated in Theorem 5.7), well-typedness is not necessary for DFI.

By design, our analysis is control-insensitive—it does not track implicit flows. In many applications, implicit flows are of serious concern. It remains possible to extend our analysis to account for such flows, following the ideas of [51, 56, 39, 37]. However, we believe that it is more practical to enforce a weaker property like DFI at the level of an operating system, and enforce stronger, control-sensitive properties like noninterference at the level of the application, with specific assumptions.

Our core security calculus is simplified, although we take care to include all aspects that require conceptual modeling for reasoning about DFI. In particular, we model threads, mutable references, binaries, and data and code pointers; other features of x86 binaries, such as recursion, control flow, and parameterized procedures, can be encoded in the core calculus. We also model all details of Windows Vista that are relevant for mandatory integrity control with dynamic labels. On the other hand, we do not model details such as discretionary access control, file virtualization, and secure authorization of privilege escalation [32], which can improve the precision of our analysis. Building a typechecker that works at the level of x86 binaries and handles all details of Windows Vista requires more work. At the same time, we believe that our analysis can be applied to more concrete programming models by translation.

Our work is closely related to that of Tse and Zdancewic [49] and Zheng and Myers [61] on noninterference in lambda calculi with dynamic security levels. While Tse and Zdancewic do not

consider mutable references in their language, it is possible to encode the sequential fragment of our calculus in the language of Zheng and Myers; however, well-typed programs in that fragment that rely on access control for DFI do not remain well-typed via such an encoding. Specifically, any restrictive access check for integrity in the presence of dynamically changing labels seems to let the adversary influence trusted computations in their system, violating noninterference [60].

Noninterference is known to be problematic for concurrent languages. In this context, Zdancewic and Myers study the notion of observational determinism [58]; Abadi, Hennessy and Riely, and others study information flow using testing equivalence [1, 29]; and Boudol and Castellani, Honda and Yoshida, and others use stronger notions based on observational equivalence [10, 30]. Sophisticated techniques that involve linearity, race analysis, behavior types, and liveness analysis also appear in the literature [30, 58, 29, 33]. While most of these techniques are developed in the setting of the pi calculus, other works consider distributed and higher-order settings to study mobile code [28, 55, 46] (as in this work).

DFI being a safety property [7] gets around some of the difficulties posed by noninterference. A related approach guides the design of the operating systems Asbestos [20] and HiStar [59], and dates back to the Clark-Wilson approach to security in commercial computer systems [16, 47]. In comparison with generic models of trace-based integrity that appear in protocol analysis, such as correspondence assertions [26, 23], our integrity model is far more specialized; as a consequence, our type system requires far less annotations than type systems for proving correspondence assertions.

Our definition of DFI relies on an operational semantics based on explicit substitution. Explicit substitution, as introduced by Abadi *et al.* [4], has been primarily applied to study the correctness of abstract machines for programming languages (whose semantics rely on substitution as a rather inefficient meta-operation), and in proof environments. It also appears in the applied pi calculus [5] to facilitate an elegant formulation of indistinguishability for security analysis. However, we seem to be the first to use explicit substitutions to track explicit flows in a concurrent language. Previously, dependency analysis [36, 6] has been applied to information-flow analysis [2, 42, 57]. These analyses track stronger dependencies than those induced by explicit flows; in particular, the dependencies are sensitive to control flows. In contrast, the use of explicit substitutions to track explicit flows seems rather obvious and appropriate in hindsight. We believe that this technique should be useful in other contexts as well.

Our analysis manifests a genuine interplay between static typing and dynamic access control for runtime protection. We seem to be the first to study this interaction in a concurrent system with dynamic labels for multi-level integrity. This approach of combining static and dynamic protection mechanisms is reflected in previous work on typing, *e.g.*, for noninterference in a Java-like language with stack inspection and other extensions [8, 41], for noninterference in lambda calculi with runtime principals and dynamic labels [49, 61], and for secrecy in concurrent storage calculi with discretionary access control mechanisms [14, 13]. A verification technique based on this approach is developed by Flanagan [22] for a lambda calculus with arbitrary base refinement types. In these studies and ours, dynamic checks complement static analysis where possible or as required, so that safety violations that are not caught statically are always caught at runtime. Moreover, static typing sometimes subsumes certain dynamic checks (as in our analysis), suggesting sound runtime optimizations. This approach is reflected in previous work on static access control [29, 43, 31].

In most real-world systems, striking the right balance between security and practice is a delicate task that is never far from controversy. It is reassuring to discover that perhaps, such a balance can be enforced formally in a contemporary operating system, and possibly improved in future ones.

Acknowledgments We wish to thank Martín Abadi, Steve Zdancewic, Pavol Černý, and several anonymous reviewers for their comments on an earlier draft of this paper. We also wish to thank John Lambert, Andrew Roths, Lantian Zheng, Karthik Bhargavan, and Cormac Flanagan for various discussions on this work.

Avik Chaudhuri's work was partly supported by Microsoft Research India, and by the National Science Foundation under Grants CCR-0208800 and CCF-0524078.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL'99: Principles of Programming Languages*, pages 147–160. ACM, 1999.

[3] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL'02: Principles of Programming Languages*, pages 33–44. ACM, 2002.

[4] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *POPL'90: Principles of Programming Languages*, pages 31–46. ACM, 1990.

[5] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL'01: Principles of Programming Languages*, pages 104–115. ACM, 2001.

[6] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *ICFP'96: Functional Programming*, pages 83–91. ACM, 1996.

[7] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(5):181–185, 1985.

[8] A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *CSFW'03: Computer Security Foundations Workshop*, pages 155–169. IEEE, 2003.

[9] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corporation, 1977.

[10] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

[11] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[12] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI'06: Operating Systems Design and Implementation*, pages 147–160. USENIX, 2006.

[13] A. Chaudhuri. Dynamic access control in a concurrent object calculus. In *CONCUR'06: Concurrency Theory*, pages 263–278. Springer, 2006.

[14] A. Chaudhuri and M. Abadi. Secrecy by typing and file-access control. In *CSFW'06: Computer Security Foundations Workshop*, pages 112–123. IEEE, 2006.

[15] A. Chaudhuri, P. Naldurg, and S. Rajamani. A type system for data-flow integrity on Windows Vista. Technical Report TR-2007-86, Microsoft Research, 2007. Also available as an arXiv e-print at `http://arxiv.org/abs/0803.3230`.

[16] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *SP'87: Symposium on Security and Privacy*, pages 184–194. IEEE, 1987.

[17] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA'07: International Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007.

[18] M. Conover. Analysis of the Windows Vista security model. Available at `www.symantec.com/avcenter/reference/Windows_`

Vista_Security_Model_Analysis.pdf.

[19] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[20] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05: Symposium on Operating Systems Principles*, pages 17–30. ACM, 2005.

[21] M. Felleisen. The theory and practice of first-class prompts. In *POPL'88: Principles of Programming Languages*, pages 180–190. ACM, 1988.

[22] C. Flanagan. Hybrid type checking. In *POPL'06: Principles of Programming Languages*, pages 245–256. ACM, 2006.

[23] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *ESOP'05: European Symposium on Programming*, pages 141–156. Springer, 2005.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *SP'82: Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[25] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *HLCL'98: High-Level Concurrent Languages*, pages 248–264. Elsevier, 1998.

[26] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[27] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR'05: Concurrency Theory*, pages 186–201. Springer, 2005.

[28] M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: A language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

[29] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.

[30] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *POPL'02: Principles of Programming Languages*, pages 81–92. ACM, 2002.

[31] D. Hoshina, E. Sumii, and A. Yonezawa. A typed process calculus for fine-grained resource access control in distributed computation. In *TACS'01: Theoretical Aspects of Computer Software*, pages 64–81. Springer, 2001.

[32] M. Howard and D. LeBlanc. *Writing Secure Code for Windows Vista*. Microsoft Press, 2007.

[33] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[34] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[35] B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, Jan 1974.

[36] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris 7, 1978.

[37] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL'05: Principles of Programming Languages*, pages 158–170. ACM, 2005.

[38] L.Wall, T.Christiansen, and R.Schwartz. *Programming Perl*. O'Reilly, 1996.

[39] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.

[40] G. C. Necula. Proof-carrying code. In *POPL'97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[41] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *SP'07: Symposium on Security and Privacy*, pages 149–163. IEEE, 2007.

[42] F. Pottier and S. Conchon. Information flow inference for free. In *ICFP'00: Functional Programming*, pages 46–57. ACM, 2000.

[43] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.

[44] M. Russinovich. *Inside Windows Vista User Access Control*. Microsoft Technet Magazine, June 2007. Available at http://www.microsoft.com/technet/technetmag/issues/2007/06/UAC/.

[45] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[46] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS'07: Logic in Computer Science*, pages 293–302. IEEE, 2007.

[47] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS'06: Network and Distributed System Security Symposium*. ISOC, 2006.

[48] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04: Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM, 2004.

[49] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *SP'04: Symposium on Security and Privacy*, pages 179–193. IEEE, 2004.

[50] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS'07: Network and Distributed System Security Symposium*. ISOC, 2007.

[51] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[52] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Scheme'07: Workshop on Scheme and Functional Programming*, 2007.

[53] Windows Vista TechCenter. *Understanding and configuring User Account Control in Windows Vista*. Available at http://technet.microsoft.com/en-us/windowsvista/aa905117.aspx.

[54] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS'07: Computer and Communications Security*, pages 116–127. ACM, 2007.

[55] N. Yoshida. Channel dependent types for higher-order mobile processes. In *POPL'04: Principles of Programming Languages*, pages 147–160. ACM, 2004.

[56] S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW'01: Computer Security Foundations Workshop*, pages 5–16. IEEE, 2001.

[57] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2/3):209–234, 2002.

[58] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW'03: Computer Security Foundations Workshop*, pages 29–43. IEEE, 2003.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI'06: Operating Systems Design and Implementation*, pages 19–19. USENIX, 2006.

[60] L. Zheng. Personal communication, July 2007.

[61] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *FAST'04: Formal Aspects in Security and Trust*, pages 27–40. Springer, 2004.