# Technical Report: The Meaning of Dynamic Integrity Taint Analysis

Sepehr Amir-Mohammadian Christian Skalka

October 24, 2016

## Abstract

Dynamic integrity taint analysis tracks explicit information flow in programs. While semantics of information flow such as noninterference are important to its understanding and implementation, nearly all formal accounts of taint analysis provide only operational definitions. An exception is the recently proposed property of *explicit secrecy* that applies to explicit flow of confidential data in low-level languages with declassification. In this work we propose a general semantic notion of explicit flow of information integrity in a high-level (Object Oriented) language with sanitization, called *explicit integrity*. The proposed semantics enables a formulation of soundness and completeness of dynamic integrity taint analysis. We study how to implement sound taint propagation mechanisms that enforce direct integrity flow policies in the presence of input sanitization according to our proposed semantics. We also discuss how completeness relies on, and has the potential to inform, implementation decisions.

# Contents

# List of Figures

# Chapter 1

# Introduction

Taint analysis has many applications [1, 2, 3], but its security semantics is not well understood. Taint analysis is typically defined as a direct, aka explicit, information flow variant, where indirect flow of information is ignored. The semantics of information flow [4] has been well studied and is typically characterized via noninterference properties [5], but surprisingly little work has been done to develop similar properties for taint analysis. Formal definitions of taint analysis implementations do exist, but they are usually operational in nature [6, 7].

An exception to this rule is work on explicit secrecy [8], which explores the semantics of dynamic confidentiality taint analysis. However this previous work only considers taint analysis for low level languages, and only with respect to data *confidentiality*. This misses a large and existing application space for dynamic *integrity* taint analysis, which is defense against injection attacks in high level code such as web applications [1, 9]. The issue here is ensuring that low integrity data sources do not directly affect high integrity sinks such as security sensitive operations.

In related work we have considered formal specifications of taint analysis for hardening Java programs against injection attacks, with a particular interest in hardening a web-based medical records software system [10]. The formulation studied here provides a meaningful semantics for the dynamic taint analysis we defined there, which is a "core" formulation of Phosphor taint analysis [11, 12]. However, in addition to Phosphor, our approach provides a semantic foundation for reasoning about other existing integrity dynamic taint analysis in HLLs [13], as well as low level languages.

## 1.1 Contributions

The main technical contribution of this work is a definition of the security property enforced by dynamic integrity taint analysis. This property, called *explicit integrity*, is general in that it applies to both high and low level languages. Our work is based on previous work on explicit secrecy [8] for low level languages. However, while information integrity can often be expressed as the dual of confidentiality [14], there is a more fundamental difference between explicit secrecy and security concerns for integrity taint analysis in HLLs. In particular, explicit secrecy is framed in terms of the attacker's ability to guess confidential values in program memory. In contrast, explicit integrity is concerned with low integrity code attacks that can be introduced, and whether they can affect high integrity events. Explicit integrity comports with applications of dynamic taint analysis to harden programs against injection attacks– where low integrity user data may maliciously taint arguments to security sensitive operations, e.g. database interactions in the case of SQL injection.

In addition to the basic definition, we also extend our notion of explicit integrity to support *endorsement*, which allows dynamic "untainting" of data usually as a result of sanitization [15, 16]. Endorsement is usually considered the integrity dual of confidentiality declassification, and we are able to adapt techniques for the latter formulated for explicit secrecy.

Aside from providing general semantic insights and formal foundations for dynamic integrity taint analyses, we also show how explicit integrity provides guiding principles for efficiency strategies in taint analysis instrumentation. In particular, system designers often treat library methods as black boxes with respect to taint propagation, i.e. taint properties are directly ascribed to library method results [17, 18]. For example, some systems will not ascribe taint to the result of character projection from tainted strings [17], and other systems will ascribe taint to the results if arguments are tainted [18], regardless of direct flow internal to the methods. The former is an example of "undertainting", while the latter is an example of "overtainting". Our formal account provides a precise characterization of these informal notions (see Definitions 3.7.2 and 3.7.3), and a basis for designers to understand the effects of implementation decisions with respect to a higher level security property.

## 1.2 Outline of the Report

In Chapter 2, we define the semantic framework for dynamic integrity taint analysis. We then briefly show how this model can be instantiated for a core imperative language. In Chapter 3, our high level OO language model is introduced. We instantiate the semantic framework and specify taint analysis logically for the language model. We also introduce required properties for user-defined taint propagation policies that could avoid incorrect implementations. This Chapter also contains examples in Section 3.8 illustrating the main ideas of our formulations, in particular incomparability of noninterference with explicit integrity. Chapter 4 describes an implementation of dynamic integrity taint analysis as a program rewriting algorithm. In Chapter 5, soundness (Definition 5.0.2) and completeness (Definition 5.0.3) of dynamic integrity taint analysis is defined and proven correct for the rewriting algorithm (Theorem 5.3.1 and Theorem 5.3.2). Related work is given in Chapter 6, and Chapter 7 concludes the report.

# Chapter 2

# Direct Integrity Semantics: Explicit Integrity

In this Chapter, we build “explicit integrity” a semantic property that builds on the intuition of *explicit secrecy* [8] and notion of *attacker power* [16]. Similar to explicit secrecy, explicit integrity is language-agnostic. In later sections, we discuss instantiation of this model for different language models.

*Explicit secrecy* is defined as a property of a program, where the execution does not change the knowledge of low confidentiality user. *Knowledge* [19] is defined as the set of initial states that a low confidentiality user is able to consider to generate a given sequence of observables. *Explicit knowledge* [8] restricts attacker knowledge for direct confidentiality flows only. Explicit knowledge cannot be used for direct integrity flow analysis however by being simply “dualized”, as it does not capture the attacker model. In the case of information integrity the attacker is an active force that attempts to inject values into the system to manipulate high integrity assets. This is different from the flow of information confidentiality, where the attacks are considered passive, i.e., they intend to gain knowledge on secret data without necessarily being able to input data.

*Attacker power* [16] is introduced as a counterpart to attacker knowledge in the context of integrity, as the set of possible low integrity inputs (attacks) that generate the same sequence of high integrity events. Each high integrity event could be a simple assignment to a predefined high integrity variable, a method that manipulates trusted data (secure sinks), etc. according to the language model. The more refined the attacker power is, the more powerful the low integrity attacker becomes, as she becomes more capable to distinguish between the effects of different attacks on high integrity data.

We define *explicit attacker power* as the attacker power constrained on direct integrity flows. Then, *explicit integrity* is defined as the property of preserving explicit attacker power during program execution. In order to limit flows to direct ones, we have followed the techniques introduced in [8] to define *state transformers*. State transformers extract direct flows semantically by specifying the ways in which program state is modified in each step of execution, along with direct-flow events that are generated. We extend state transformers for integrity flows by specifying the role of low integrity inputs in state modifications and events. Moreover, we generalize the state transformer definition in a way that supports both imperative and functional language instantiations.

## 2.1 The Threat Model

We assume that programs are trusted to not be malicious, but potentially erroneous. This assumption is important for taint analysis as it usually assumes that indirect flows cannot be used as a covert channel. Programs are executed in trusted environments. We assume that untrusted data sources provide low integrity input that could manipulate high integrity sinks. These inputs are conditioned to be *fair*, i.e. low integrity input cannot affect trusted sinks directly. Moreover, low integrity users have access to program code. We also assume that our enforcement mechanism (taint analysis instrumentation) is non-malicious.

## 2.2 Model Specification

In what follows, we formulate our explicit integrity semantics. We first define the interface for our framework. Let $\mathbf{K}$, $\mathbf{C}$, $\mathbf{S}$ and $\mathbf{E}$ be the sets of program configurations, controls, states and high integrity events, respectively such that $\kappa \in \mathbf{K}$, $\mathrm{c} \in \mathbf{C}$, $\mathrm{s} \in \mathbf{S}$ and $\mathrm{e} \in \mathbf{E}$. We assume the existence of the small step evaluation relation $\rightarrow \subseteq \mathbf{K} \times \mathbf{E}^* \times \mathbf{K}$ where $(\kappa, \bar{\mathrm{e}}, \kappa') \in \rightarrow$ is denoted as $\kappa \xrightarrow{\bar{\mathrm{e}}} \kappa'$. We use $\kappa \rightarrow \kappa'$ if $\bar{\mathrm{e}}$ is empty ($\epsilon$) or insignificant. Notation $\rightarrow^*$ is used for reflexive and transitive closure of $\rightarrow$.

Each configuration is considered to include two segments: control (code) and state (data). These segments are not necessarily disjoint and could overlap in some language models. In this regard, let mappings $\mathit{state} : \mathbf{K} \rightarrow \mathbf{S}$ and $\mathit{control} : \mathbf{K} \rightarrow \mathbf{C}$ extract the state and control segments of configurations, and $\langle \cdot, \cdot \rangle : \mathbf{C} \times \mathbf{S} \rightarrow \mathbf{K}$ construct a configuration from its control and state segments. We posit a preorder binary relation on $\mathbf{C}$ that represents compatibility of one control with another, denoted by $\mathrm{c} \preccurlyeq \mathrm{c}'$. This relation needs to be instantiated for the object language.

We assume the existence of an entry point $[\cdot]$ in the controls denoted by $\mathrm{c}[\cdot]$ by which the attacker can inject low integrity input. The attacker code is denoted by $\mathrm{a}$. Then $\mathrm{c}[\mathrm{a}]$ represents a control in which the attacker has injected code $\mathrm{a}$. Note that an attack $\mathrm{a}$ is a control itself, i.e., $\mathrm{a} \in \mathbf{C}$. An attack is called fair provided that whenever executed, no high integrity events are generated, i.e., the attack does not manipulate high integrity data by itself.

**Definition 2.2.1** *An attack* $\mathrm{a}$ *is* fair *provided that for all* $\bar{\mathrm{e}}$, $\mathrm{s}$ *and* $\kappa$*, if* $\langle \mathrm{a}, \mathrm{s} \rangle \xrightarrow{\bar{\mathrm{e}}}{}^* \kappa$ *then* $\bar{\mathrm{e}} = \epsilon$.

We define state transformers by extending the definition in [8] to consider low integrity user inputs and predicating on compatible controls. The latter provides a unified semantic framework for both imperative and functional language settings, by giving a more permissive definition of state transformers than in [8].

**Definition 2.2.2** *Let $\kappa \rightarrow \kappa'$ and $control(\kappa) = \mathrm{c}[\mathrm{a}]$. The* state transformer $f : \mathbf{C} \times \mathbf{S} \rightarrow \mathbf{S} \times \mathbf{E}^*$ *is the function where $f(\mathrm{a}', \mathrm{s}) = (state(\kappa''), \bar{\mathrm{e}})$ for the unique $\kappa''$ and $\bar{\mathrm{e}}$ such that $\langle \mathrm{c}'[\mathrm{a}'], \mathrm{s}\rangle \xrightarrow{\bar{\mathrm{e}}} \kappa''$ for all $\mathrm{c}' \preccurlyeq \mathrm{c}$. We write $\kappa \rightarrow_f \kappa'$ to associate the state transformer $f$ with the reduction $\kappa \rightarrow_f \kappa'$.*

This definition is then extended to multiple evaluation steps by composing state transformers at each step. Let $f(\mathrm{a}, \mathrm{s}) = (\mathrm{s}', \bar{\mathrm{e}})$ and $g(\mathrm{a}', \mathrm{s}') = (\mathrm{s}'', \bar{\mathrm{e}}')$. Then, $(g * f)(\mathrm{a}, \mathrm{s}) = (\mathrm{s}'', \bar{\mathrm{e}}\,\bar{\mathrm{e}}')$.

We now define the power an attacker obtains by observing high integrity events. We capture this by defining a set of possible attacks that the attacker can launch so that they generate the same sequence of high integrity events. For an attack $\mathrm{a}$, a state $\mathrm{s}$ and some state transformer $f$, an attack $\mathrm{a}'$ is considered possible if it agrees on the generated integrity events with $\mathrm{a}$.

**Definition 2.2.3** *We define* explicit attacker power *with respect to attack $\mathrm{a}$, state $\mathrm{s}$ and state transformer $f$ as follows, where projection on the $i$th element of a tuple is denoted by $\pi_i$.*

$$p_e(\mathrm{a}, \mathrm{s}, f) = \{\mathrm{a}' \mid \pi_2(f(\mathrm{a}, \mathrm{s})) = \pi_2(f(\mathrm{a}', \mathrm{s}))\}.$$

A control then satisfies explicit integrity for some state iff no attack can be excluded from observing the high integrity events generated by the extracted state transformer.

**Definition 2.2.4** *A control $\mathrm{c}[\cdot]$ satisfies* explicit integrity *for state $\mathrm{s}$, iff $\langle \mathrm{c}[\mathrm{a}], \mathrm{s}\rangle \rightarrow_f^* \kappa'$ implies that for any $\mathrm{s}'$, $\mathrm{a}'$ and $\mathrm{a}''$, $\mathrm{a}'' \in p_e(\mathrm{a}', \mathrm{s}', f)$. A control $\mathrm{c}[\cdot]$ satisfies* explicit integrity *iff for any $\mathrm{s}$, $\mathrm{c}[\cdot]$ satisfies explicit integrity for $\mathrm{s}$.*

We can now consider explicit integrity in the presence of endorsement in the style of gradual release [19]. We assume that there exists a set of integrity events $\mathbf{E_{en}} \subseteq \mathbf{E}$ that are generated when endorsements occur. Explicit attacker power is only allowed to change for such events.

**Definition 2.2.5** *A control* $\mathrm{c}[\cdot]$ *satisfies* explicit integrity modulo endorsement *for state* $\mathrm{s}$ *iff* $\langle \mathrm{c}[\mathrm{a}], \mathrm{s}\rangle \rightarrow_f^* \kappa' \xrightarrow{\bar{\mathrm{e}}}{}_g^* \kappa''$ *and* $\bar{\mathrm{e}} \notin \mathbf{E_{en}}^*$ *imply that* $p_e(\mathrm{a}', \mathrm{s}, f) = p_e(\mathrm{a}', \mathrm{s}, g * f)$ *for any attack* $\mathrm{a}'$.

We also define a variant of noninterference for the sake of comparison with explicit integrity. In this regard, a particular binary relation between states, called *trust equivalence*, is posited and denoted by $\mathrm{s} =_{\mathbb{T}} \mathrm{s}'$.

**Definition 2.2.6** *Program* $\mathrm{c}[\cdot]$ *is noninterfering iff for any two states* $\mathrm{s}$ *and* $\mathrm{s}'$ *and attacks* $\mathrm{a}$ *and* $\mathrm{a}'$, *if* $\langle \mathrm{c}[\mathrm{a}], \mathrm{s}\rangle \xrightarrow{\bar{\mathrm{e}}}{}^* \langle \epsilon, s^*\rangle$, $\langle \mathrm{c}[\mathrm{a}'], \mathrm{s}'\rangle \xrightarrow{\bar{\mathrm{e}}'}{}^* \langle \epsilon, s'^*\rangle$ *and* $\mathrm{s} =_{\mathbb{T}} \mathrm{s}'$, *we have* $\bar{\mathrm{e}} = \bar{\mathrm{e}}'$.

An instantiation of the model for a core procedural language with mutable references is given is the following. In Section 3.5, we demonstrate how this property could be studied in a functional setting.

## 2.3 Model Instantiation for Imperative Settings

In this section, we give a brief description of direct integrity flows for imperative languages. Similar to previous work [16], we define commands with entry points for user code injection, and define attacks as commands. The details are given in Figure 2.1.

Note that attacks are assumed to be fair, so they do not include assignments with high integrity variable ($\mathbb{T}$) to be on the left hand side. Let $u$ and $t$ range over variables such that $\Gamma(u) = \mathbb{U}$ and $\Gamma(t) = \mathbb{T}$, i.e, $u$ and $t$ are low integrity (untrusted) and high integrity (trusted) variables, respectively. We assume the standard operational semantics for the language, except for assignments in which the left hand side is some high integrity variable $t$. This case represents a high integrity event, in which the operational semantics also generates an event in the form of $(t, v)$. Moreover, endorsing an expression generates an event of the form $en(e)$.

| | |
|---|---|
| $x \in \mathit{Var}$ | Variables |
| $m \in \mathit{Mem}$ | Memory |
| $\mathit{Mem} = \mathit{Var} \to \mathit{Int}$ | Memories |
| $e ::= n \mid x \mid e\ \mathit{op}\ e$ | Expressions |
| $\mathrm{a}, c ::= \mathbf{skip} \mid x := e \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c; c \mid x := \mathbf{endorse}(e)$ | Commands (Attacks) |
| $c[\cdot] ::= c \mid [\cdot]$ | Commands with entry points |
| $\kappa := (c, m)$ | Configurations |
| $\Gamma = \mathit{Var} \to \{\mathbb{T}, \mathbb{U}\}$ | Ingterity level |

Figure 2.1: The core imperative calculus.

We now define the interface for explicit integrity. Let $\mathbf{K}$ be the set of $\kappa$ defined in Figure 2.1. $\mathbf{C}$ is the set of commands, and $\mathbf{S}$ is $\mathit{Mem}$. $\mathbf{E}$ is the set of events of the form $(t, v)$ and $en(e)$. Mappings $\mathit{control}$ and $\mathit{state}$ are simple projection functions over pairs, and $\langle \cdot, \cdot \rangle$ is pairing function. Compatibility relation on controls is defined as equality, i.e., for any command $c$, $c \preccurlyeq c$. State transformer $f$ can be defined according to the operational semantics. In particular for high integrity events, $f(\mathrm{a}, m) = (m[t \mapsto v], [(t, v)])$, when command $t := e$ is being executed and $(e, m) \downarrow v$.[1]

Also, we define two memories trust equivalent $m =_{\mathbb{T}} m'$ iff for any variable $x$, if $\Gamma(x) = \mathbb{T}$ then $m(x) = m'(x)$.

## 2.3.1 Examples

Here we give two examples that demonstrate the incomparability of noninterference with explicit integrity.

**Example 2.3.1** *Let* $c[\cdot] = [\cdot]; \mathbf{if}\ u = 0\ \mathbf{then}\ t := 1\ \mathbf{else}\ t := 2$. *This program is interfering as the value of* $u$ *is implicitly flowed to* $t$. *For example by attacks* $\mathrm{a} = (u := 0)$ *and* $\mathrm{a}' = (u := 1)$ *and any two trust equivalent memories the program generates two different sequence of high*

[1]We assume big step semantics to evaluate expressions similar to previous work [8, 19, 16].

*integrity events:* $[(t, 1)]$ *and* $[(t, 2)]$*. However, it satisfies explicit integrity because there are no explicit flows from* $u$ *to* $t$*. More formally, the state transformer would be* $f(\mathrm{a}, m) = (m[u \mapsto v][t \mapsto i], [(t, i)])$*, for* $i = 1, 2$ *depending on what value* $v$ *attack* $\mathrm{a}$ *assigns to* $u$*. Then for any attack* $\mathrm{a}'$ *and memory* $m$*, the explicit attacker power is preserved, since* $\pi_2(f(\mathrm{a}, m)) = \pi_2(f(\mathrm{a}', m)) = [(t, i)]$*.*

**Example 2.3.2** *Let* $c[\cdot] = [\cdot]; \mathbf{if}\ u = 0\ \mathbf{then}\ t := u\ \mathbf{else}\ t := 0$*. This program is obviously noninterfering, since independent of what is assigned to* $u$ *through attacks the same sequence of high integrity events is generated:* $[(t, 0)]$*. But it does not satisfy explicit integrity as the information is propagated explicitly from* $u$ *to* $t$*. For example for some attack* $\mathrm{a}$ *that assigns 0 to* $u$*, the state transformer is defined as* $f(\mathrm{a}, m) = (m[t \mapsto m(u)], [(t, m(u))])$*. Then, for some attack* $\mathrm{a}' = (u := 1)$*, the explicit attacker power is refined to the set of all attacks that in some way assign 1 to* $u$*, i.e.,* $p_e(\mathrm{a}', m, f) \neq p_e(\mathrm{a}', m, g)$ *in Definition 2.2.4.*

# Chapter 3

# An OO Model

For practical purposes we are interested in applications of taint analysis for HLLs, especially Java [10]. Therefore in our formulation we consider a core model of Java, which is based on Featherweight Java (FJ) [20]. FJ is a core calculus that includes class hierarchy definitions, subtyping, dynamic dispatch, and other basic features of Java. To this we add semantics for library methods that allow specification of operations on base values (such as strings and integers). Consideration of these features is important for a thorough modeling of Phosphor-style taint analysis, and important related issues such as string- vs. character-based taint [17] which have not been considered in previous formal work on taint analysis [6]. Since static analysis is not a topic of this work, for brevity we omit the standard FJ type analysis which is described in [20].

## 3.1 Syntax and Semantics

The syntax and semantics of FJ is defined in Figure 3.1. We let $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}$ range over class names, $\mathtt{x}$ range over variables, $\mathtt{f}$ range over field names, and $\mathtt{m}$ range over method names. *Values*, denoted $\mathtt{v}$ or $\mathtt{u}$, are objects, i.e. expressions of the form $\mathtt{new}\ \mathtt{C}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$. We assume

$$\mathtt{L} ::= \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{K} ::= \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} \qquad \mathtt{M} ::= \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\}$$

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid \mathtt{C.m}(\mathtt{e}) \qquad \mathtt{E} ::= [\,] \mid \mathtt{E.f} \mid \mathtt{E.m}(\bar{\mathtt{e}}) \mid \mathtt{v.m}(\bar{\mathtt{v}}, \mathtt{E}, \bar{\mathtt{e}}') \mid \mathtt{new\ C}(\bar{\mathtt{v}}, \mathtt{E}, \bar{\mathtt{e}}') \mid \mathtt{C.m}(\mathtt{E})$$

$$\mathit{fields}_{CT}(\mathtt{Object}) = \varnothing \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathit{fields}_{CT}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}}{\mathit{fields}_{CT}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}, \bar{\mathtt{C}}\ \bar{\mathtt{f}}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{B\ m}(\bar{\mathtt{B}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} \in \bar{\mathtt{M}}}{\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}, \mathtt{e}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \notin \bar{\mathtt{M}}}{\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \mathit{mbody}_{CT}(\mathtt{m}, \mathtt{D})}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{B\ m}(\bar{\mathtt{B}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} \in \bar{\mathtt{M}}}{\mathit{mtype}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{B}} \to \mathtt{B}} \qquad \frac{\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \notin \bar{\mathtt{M}}}{\mathit{mtype}_{CT}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}_{CT}(\mathtt{m}, \mathtt{D})}$$

$$\text{Context}\ \frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{E}[\mathtt{e}] \to \mathtt{E}[\mathtt{e}']} \qquad \text{Field}\ \frac{\mathit{fields}_{CT}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \qquad \mathtt{f}_i \in \bar{\mathtt{f}}}{\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{f}_i \to \mathtt{v}_i} \qquad \text{Invoke}\ \frac{\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}, \mathtt{e}}{\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}) \to \mathtt{C.m}(\mathtt{e}[\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}][\bar{\mathtt{u}}/\bar{\mathtt{x}}])} \qquad \text{Return}\ \mathtt{C.m}(\mathtt{v}) \to \mathtt{v}$$

Figure 3.1: FJ Syntax and Semantics

given an `Object` value that has no fields or methods. In addition to the standard expressions of FJ, we introduce a new form $\mathtt{C.m}(\mathtt{e})$. This form is used to identify the method $\mathtt{C.m}$ associated with a current evaluation context (aka the "activation frame"). This does not really change the semantics, but is a useful feature for our specification of sanitizer endorsement since return values from sanitizers need to be endorsed– see the Invoke and Return rules in the operational semantics below for its usage.

For brevity in this syntax, we use vector notations. Specifically we write $\bar{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$, similarly for $\bar{\mathtt{C}}$, $\bar{\mathtt{m}}$, $\bar{\mathtt{x}}$, $\bar{\mathtt{e}}$, etc., and we write $\bar{\mathtt{M}}$ as shorthand for $\mathtt{M}_1 \cdots \mathtt{M}_n$. We write the empty sequence as $\varnothing$, we use a comma as a sequence concatenation operator. If and only if $\mathtt{m}$ is one of the names in $\bar{\mathtt{m}}$, we write $\mathtt{m} \in \bar{\mathtt{m}}$. Vector notation is also used to abbreviate sequences of declarations; we let $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$ and $\bar{\mathtt{C}}\ \bar{\mathtt{f}};$ denote $\mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n$ and $\mathtt{C}_1\ \mathtt{f}_1; \ldots; \mathtt{C}_n\ \mathtt{f}_n;$

respectively. The notation $\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}}$; abbreviates $\mathtt{this.f}_1 = \mathtt{f}_1; \ldots; \mathtt{this.f}_n = \mathtt{f}_n$;. Sequences of names and declarations are assumed to contain no duplicate names.

### 3.1.1 The class table and field and method body lookup

The class table $CT$ maintains class definitions. The manner in which we look up field and method definitions implements inheritance and override, which allows fields and methods to be redefined in subclasses. We assume a given class table $CT$ during evaluation, which will be clear from context.

### 3.1.2 Method type lookup

Just as we've defined a function for looking up method bodies in the class table, we also define a function that will look up method types in a class table. Although we omit FJ type analysis from this presentation, method type lookup will be useful for the constraints specified for endorsement (Section 3.4).

### 3.1.3 Operational semantics

We define the operational semantics of FJ as a small step relation in the usual manner.

We use $\to^*$ to denote the reflexive, transitive closure of $\to$. We will also use the notion of an *execution trace* $\tau$ to represent a series of *configurations* $\kappa$, where $\tau = \kappa_1 \ldots \kappa_n$ means that $\kappa_i \to \kappa_{i+1}$ for $0 < i < n$. In the case of FJ, configurations $\kappa$ are just expressions $\mathtt{e}$. Note that an execution trace $\tau$ may represent the partial execution of a program, i.e. the trace $\tau$ may be extended with additional configurations as the program continues execution. We use metavariables $\tau$ and $\sigma$ to range over traces.

To denote execution of top-level programs $\mathtt{p}(\mathrm{a})$ where $\mathrm{a}$ is an object of low integrity, we assume that all class tables $CT$ include an entry point `TopLevel.main`, where `TopLevel` objects have no fields. We define $\mathfrak{p}(\mathrm{a}) = \mathtt{new\ TopLevel().main}(\mathrm{a})$, and we write $\mathfrak{p}(\mathrm{a}) \Downarrow \tau$ iff trace $\tau$ begins with the configuration $\mathfrak{p}(\mathrm{a})$.

## 3.2 Library Methods

The abstract calculus described above is not particularly interesting with respect to direct information flow and integrity propagation, especially since method dispatch is considered an indirect flow. More interesting is the manner in which taint propagates through primitive values and library operations on them, especially strings and string operations. This is because direct flows should propagate through some of these methods. Also, for run-time efficiency and ease of coding some Java taint analysis tools treat even complex library methods as "black boxes" that are instrumented at the top level for efficiency [18], rather than relying on instrumentation of lower-level operations.

Note that treating library methods as "black boxes" introduces a potential for over- and under-tainting– for example in some systems all string library methods that return strings are instrumented to return tainted results if any of the arguments are tainted, regardless of any direct flow from the argument to result [18]. Clearly this strategy introduces a potential for over-taint. Other systems do not propagate taint from strings to their component characters when decomposed [17], which is an example of under-taint. Part of our goal here is to develop an adequate language model to consider these approaches.

We therefore extend our basic definitions to accommodate primitive values and their manipulation. Let a *primitive field* be a field containing a primitive value. We call a *primitive object/class* any object/class with primitive fields only, and a *library method* is any method that operates on primitive objects, defined in a primitive class. We expect primitive objects to be object wrappers for primitive values (e.g., $\mathtt{Int}(5)$ wrapping primitive value $\mathtt{5}$), and library methods to be object-oriented wrappers over primitive operations (e.g., $\mathtt{Int\ plus(Int)}$

wrapping primitive operation $+$), allowing the latter's embedding in FJ. As a sanity condition we only allow library methods to select primitive fields or perform primitive operations. Let $\mathit{LibMeths}$ be the set of library method names paired with their corresponding primitive class names.

We posit a special set of field names $\mathit{PrimField}$ that access primitive values ranged over by $\nu$ that may occur in objects, and a set of operations ranged over by $\mathit{Op}$ that operate on primitive values. We require that special field name selections only occur as arguments to $\mathit{Op}$, which can easily be enforced in practice by a static analysis. Similarly, primitive values $\nu$ may only occur in special object fields and be manipulated there by any $\mathit{Op}$.

$$\begin{array}{rcl}
\mathtt{f}^* & \in & \mathit{PrimField} \\
e & ::= & \nu \mid \mathtt{e.f}^* \\
\mathtt{e} & ::= & \cdots \mid \mathit{Op}(\bar{e}) \\
\mathtt{v} & ::= & \mathtt{new\,C}(\bar{\mathtt{v}}) \mid \nu \\
\mathtt{E} & ::= & \cdots \mid \mathit{Op}(\bar{\nu}, \mathtt{E}, \bar{e})
\end{array}$$

For library methods we require that the body of any library method be of the form where C is a primitive class:return $\mathtt{new\,C}(\bar{\mathtt{e}}_1, \ldots, \bar{\mathtt{e}}_n)$. We define the meaning of operations $\mathit{Op}$ via an "immediate" big-step semantic relation $\approx$ where the rhs of the relation is required to be a primitive value, and we identify expressions up to $\approx$. For example, to define a library method for integer addition, where `Int` objects contain a primitive numeric `val`, field we would define a $+$ operation as $+(n_1, n_2) \approx n_1 + n_2$. Then we can add to the definition of `Int` in $\mathit{CT}$ a method `Plus` to support arithmetic in programs:

$$\mathtt{Int\,plus(Int\,x)}\ \{\ \mathtt{return(new(Int)(+(this.val, x.val)));}\ \}$$

Similarly, to define string concatenation, we define a concatenation operation @ on primitive strings: $@(s_1, s_2) \approx s_1 s_2$ and we extend the definition of `String` in $CT$ with the following method, where we assume all `String` objects maintain their primitive representation in a `val` field:

$$\begin{array}{l}\texttt{String concat(String x)} \\ \qquad\qquad \{\ \texttt{return(new(String)(@(this.val, x.val)));}\ \}\end{array}$$

## 3.3 The Security Model

The security problem we consider is about the integrity of data being passed to security sensitive operations (ssos). An important example is a string entered by an untrusted users that is passed to a database method for parsing and execution as a SQL command. The security mechanism should guarantee that low-integrity data cannot be passed to ssos without previous sanitization. While there are no primitive conditional expressions in our Java model, indirect flows are realized via dynamic method dispatch which faithfully models Java dispatch.

More precisely, we posit that programs $\texttt{p}(\mathrm{a})$ in this security setting contain a low integrity data source (an attack) $\mathrm{a}$, and an arbitrary number of secure sinks (ssos) and sanitizers which are specified externally to the program by a security administrator. For simplicity we assume that ssos are unary operations over primitive objects, so there is no question about which argument may be tainted. Since we define a Java based model, each sso or sanitizer is identified as a specific method `m` in a class `C`. That is, there exists a set of $\mathit{Sanitizers}$ containing class, method pairs `C.m` which are assumed to return high-integrity data, though they may be passed low-integrity data. Likewise, there exists a set of $\mathit{SSOs}$ of the same form, and for brevity we will write $\mathit{sso}(\texttt{e})$ for a method invocation `new o.m(e)` on some object `o` where $\texttt{C.m} \in \mathit{SSO}$. As a sanity condition we require $\mathit{SSOs} \cap \mathit{Sanitizers} = \varnothing$. For simplicity of our formal presentation we assume that only one tainted source will exist.

## 3.4 Endorsement

In order to discuss explicit integrity modulo endorsement for FJ, we extend the language with endorsement methods being invoked on sanitized values. In this regard, we assume the existence of $\texttt{endorse}()$ method satisfying the following constraints.

$$\frac{\texttt{C.m} \in \mathit{Sanitizers} \qquad \mathit{mtype}_{CT}(\texttt{m},\texttt{C}) = \bar{\texttt{C}} \to \texttt{D} \qquad \mathit{fields}_{CT}(\texttt{D}) = \bar{\texttt{f}}}{\mathit{mbody}_{\mathcal{R}(CT)}(\texttt{endorse},\texttt{D}) \quad = \quad \varnothing, \texttt{new D}(\overline{\texttt{this.f}})}$$

$$\frac{\texttt{C.m} \in \mathit{Sanitizers}}{\mathit{mbody}_{\mathcal{R}(CT)}(\texttt{m},\texttt{C}) = \bar{\texttt{x}}, \texttt{e.endorse}()}$$

## 3.5 Model instantiation for FJ

In this section, we instantiate explicit integrity for FJ. First, we define the required interface specified in Chapter 2. Let $\mathbf{C}$ and $\mathbf{S}$ both be the set of expressions $\texttt{e}$. We define $\mathbf{K}$ as the set of expressions as well. Mapping $\mathit{control}$ and $\mathit{state}$ are both simple identity functions over expressions and, $\langle\cdot,\cdot\rangle : (\texttt{e},\texttt{e}) \mapsto \texttt{e}$. The set $\mathbf{E}$ is the set of events of the form $\mathit{ie}(\texttt{v})$ and $\mathit{en}(\texttt{v})$. We modify the operational semantics of FJ slightly in order to consider integrity events. When an sso is invoked with argument $\texttt{v}$ the integrity event $\mathit{ie}(\texttt{v})$ is generated. Moreover, the event $\mathit{en}(\texttt{v})$ is generated when $\texttt{v.endorse}()$ is being executed. We define expression $\texttt{e}$ to be compatible with $\texttt{e}'$ if $\texttt{e}$ and $\texttt{e}'$ are shape-conformant and classes of the values in $\texttt{e}$ are subtypes ($<:$) of classes of the values in $\texttt{e}'$. The definition of compatibility is given in Figure 3.2.

We have a single code injection (attack) entry point in FJ top-level programs $\mathfrak{p}$ which we've defined to be of the form $\texttt{new TopLevel().main}([\cdot])$. Attacks are defined as values being fed to $\texttt{TopLevel.main}$, i.e., $a ::= \texttt{new C}(\bar{\texttt{v}})$.

State transformers can be defined in a straightforward way according to the reduction rules

$$\mathtt{x} \preccurlyeq \mathtt{x} \qquad \frac{\mathtt{e} \preccurlyeq \mathtt{e}'}{\mathtt{e.f} \preccurlyeq \mathtt{e}'\mathtt{.f}} \qquad \frac{\mathtt{e} \preccurlyeq \mathtt{e}' \quad \bar{\mathtt{e}} \preccurlyeq \bar{\mathtt{e}}'}{\mathtt{e.m}(\bar{\mathtt{e}}) \preccurlyeq \mathtt{e}'\mathtt{.m}(\bar{\mathtt{e}}')} \qquad \frac{\bar{\mathtt{e}} \preccurlyeq \bar{\mathtt{e}}'}{\mathtt{new\ C}(\bar{\mathtt{e}}) \preccurlyeq \mathtt{new\ C}(\bar{\mathtt{e}}')} \qquad \frac{\mathtt{C} <: \mathtt{D}}{\mathtt{new\ C}(\bar{\mathtt{v}}) \preccurlyeq \mathtt{new\ D}(\bar{\mathtt{u}})}$$

$$\frac{\mathtt{e} \preccurlyeq \mathtt{e}'}{\mathtt{C.m(e)} \preccurlyeq \mathtt{C.m}(\mathtt{e}')} \qquad \nu \preccurlyeq \nu' \qquad \frac{\mathtt{e} \preccurlyeq \mathtt{e}'}{Op(\mathtt{e}) \preccurlyeq Op(\mathtt{e}')} \qquad \mathtt{e} \preccurlyeq \mathtt{e} \qquad \frac{\mathtt{e} \preccurlyeq \mathtt{e}' \quad \mathtt{e}' \preccurlyeq \mathtt{e}''}{\mathtt{e} \preccurlyeq \mathtt{e}''}$$

Figure 3.2: Compatibility of expressions.

$$\frac{\mathtt{C.m} \in SSOs \qquad mbody_{CT}(\mathtt{m},\mathtt{C}) = \mathtt{x},\mathtt{e}}{f(\mathrm{a}, \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}})\mathtt{.m(u)}]) = (\mathtt{E}[\mathtt{C.m}(\mathtt{e}[\mathtt{u}/\mathtt{x}][\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}])], ie(\mathtt{u}))}$$

$$f(\mathrm{a}, \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}})\mathtt{.endorse}()]) = (\mathtt{E}[\mathtt{C.endorse}(\mathtt{new\ C}(\bar{\mathtt{v}}))], en(\mathtt{new\ C}(\bar{\mathtt{v}})))$$

$$\frac{mbody_{CT}(\mathtt{main}, \mathtt{TopLevel}) = \mathtt{x},\mathtt{e}}{f(\mathrm{a}, \mathtt{new\ TopLevel().main(u)}) = (\mathtt{TopLevel.main}(\mathtt{e}[\mathrm{a}/\mathtt{x}][\mathtt{new\ TopLevel()}/\mathtt{this}]), \epsilon)}$$

$$\frac{\mathtt{e} \to \mathtt{e}' \qquad \mathtt{e} \notin \{\mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}})\mathtt{.m(u)}], \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}})\mathtt{.endorse}()], \mathtt{new\ TopLevel().main(u)}\}}{f(\mathrm{a}, \mathtt{e}) = (\mathtt{e}', \epsilon)}$$

Figure 3.3: State transformer definition for each reduction step.

in FJ. They are defined as partial functions in contrast to the ones defined in the previous Chapter for an imperative calculus. The partiality comes from syntactic constraints inflicted on expressions. We only need particular cases for sso and endorsement invocations. When method `TopLevel.main` is invoked with an attack a, the attack is used as an argument. In the rest of cases, state transformers imitate reduction steps and do not generate integrity events. The definitions are given in Figure 3.3.

## 3.6 Logical Specification of Integrity Analysis

In this Section we demonstrate how direct integrity taint analysis for FJ can be expressed as a policy separate from code. To accomplish this we interpret program traces as information represented by a logical fact base in the style of Datalog [21, 22]. We then define a predicate called $\mathrm{Shadow}$ that inductively constructs a "shadow" expression that reflects the proper taint for all data in a configuration at any point in a trace.

An important feature of Java based taint analyses is that they tend to be object based, i.e. each object has an assigned taint level. In our model, a shadow expression has a syntactic structure that matches up with the configuration expression, and associates integrity levels (including "high" $\circ$ and "low" $\bullet$) with particular objects via shape conformance.

**Example 3.6.1** *Suppose a method* `m` *of an untainted* `C` *object with no fields is invoked on a pair of tainted $s_1$ and untainted $s_2$ strings:* $\texttt{new C().m(new String(}s_1\texttt{), new String(}s_2\texttt{))}$. *The proper shadow is:*

$$\texttt{shadow C}(\circ)\texttt{.m}(\texttt{shadow String}(\bullet), \texttt{shadow String}(\circ)).$$

On the basis of shadow expressions that correctly track integrity, we can logically specify taint analysis as a property of shadowed trace information.

## 3.6.1 Taint Tracking as a Logical Trace Property

We develop a mapping $toFOL(\cdot)$ that interprets FJ traces as sets of logical facts (a fact base). Intuitively, in the interpretation each configuration is represented by a $\mathrm{Context}$ predicate representing the evaluation context, and predicates (e.g. $\mathrm{Call}$) representing redexes. Each of these predicates have an initial natural number argument denoting a "timestamp", reflecting the ordering of configurations in a trace.

**Definition 3.6.1** *We define $toFOL(\cdot)$ as a mapping on traces and configurations:*

$$toFOL(\tau) = \bigcup_{\sigma \in \mathbf{prefix}(\tau)} toFOL(\sigma)$$

$$\begin{aligned}
&toFOL(\mathtt{v}, n) = \{\text{Value}(n, \mathtt{v})\},\\
&toFOL(\mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{f}], n) = \{\text{GetField}(n, \mathtt{new\ C}(\bar{\mathtt{v}}), \mathtt{f}), \text{Context}(n, \mathtt{E})\},\\
&toFOL(\mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}})], n) = \{\text{Call}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}), \text{Context}(n, \mathtt{E})\},\\
&toFOL(\mathtt{E}[\mathtt{C.m}(\mathtt{v})], n) = \{\text{ReturnValue}(n, \mathtt{C}, \mathtt{m}, \mathtt{v}), \text{Context}(n, \mathtt{E})\},\\
&toFOL(\mathtt{E}[Op(\bar{\nu})], n) = \{\text{PrimCall}(n, Op, \bar{\nu}), \text{Context}(n, \mathtt{E})\}.
\end{aligned}$$

Figure 3.4: Definition of $toFOL(\cdot)$ for configurations.

$$\begin{aligned}
&\text{match}(sv, [\,], sv).\\
&\text{match}(\mathtt{shadow\ C}(t, \overline{sv}).\mathtt{f}_i, [\,], \mathtt{shadow\ C}(t, \overline{sv}).\mathtt{f}_i).\\
&\text{match}(\mathtt{shadow\ C}(t, \overline{sv}).\mathtt{m}(\overline{su}), [\,], \mathtt{shadow\ C}(t, \overline{sv}).\mathtt{m}(\overline{su})).\\
&\text{match}(\mathtt{C.m}(sv), [\,], \mathtt{C.m}(sv)).\\
&\text{match}(se, SE, se') \implies \text{match}(se.\mathtt{f}, SE.\mathtt{f}, se').\\
&\text{match}(se, SE, se') \implies \text{match}(se.\mathtt{m}(\overline{se}), SE.\mathtt{m}(\overline{se}), se').\\
&\text{match}(se, SE, se') \implies\\
&\ \text{match}(sv.\mathtt{m}(\overline{sv}, se, \overline{se}), sv.\mathtt{m}(\overline{sv}, SE, \overline{se}), se').\\
&\text{match}(se, SE, se') \implies\\
&\ \text{match}(\mathtt{shadow\ C}(t, \overline{sv}, se, \overline{se}), \mathtt{shadow\ C}(t, \overline{sv}, SE, \overline{se}), se').\\
&\text{match}(se, SE, se') \implies \text{match}(\mathtt{C.m}(se), \mathtt{C.m}(SE), se').\\
&\text{match}(se, SE, se') \implies \text{match}(Op(\bar{sv}, se, \bar{se}), Op(\bar{sv}, SE, \bar{se}), se').
\end{aligned}$$

Figure 3.5: match predicate definition.

*such that* $toFOL(\sigma) = \bigcup_i toFOL(\kappa_i, i)$ *for* $\sigma = \kappa_1 \cdots \kappa_k$. *We define* $toFOL(\kappa, n)$ *as in Figure 3.4.*

## Integrity Identifiers

We introduce an integrity identifier $t$ that denotes the integrity level associate with objects. We assume the existence of a lattice of taint tags with at least two elements, to denote high integrity ($\circ$) and low integrity ($\bullet$) [1]. Let $\bullet$ and $\circ$ be the least and greatest elements of the lattice respectively. We specify an ordering $\leq$ on these labels denoting their integrity relation: $\bullet \leq \circ$.

[1]We use different notation for taint tags to indicate that they are dynamic labels.

We posit the usual meet $\wedge$ and join $\vee$ operations on taint lattice elements, and introduce logical predicates $\mathrm{meet}$ and $\mathrm{join}$ such that $\mathrm{meet}(t_1 \wedge t_2, t_1, t_2)$ and $\mathrm{join}(t_1 \vee t_2, t_1, t_2)$ hold.

### Shadow Traces, Taint Propagation, and Sanitization

Shadow traces reflect taint information of objects as they are passed around programs. Shadow traces manipulate shadow terms and context, which are terms $T$ in the logic with the following syntax. Note the structural conformance with closed `e` and `E`, but with primitive values replaced with a single dummy value $\delta$ that is omitted for brevity in examples, but is necessary to maintain proper arity for field selection. Shadow expressions most importantly assign integrity identifiers $t$ to objects:

$$\begin{array}{rcl}
sv & ::= & \texttt{shadow}\ \texttt{C}(t, \bar{sv}) \mid \delta \\
se & ::= & sv \mid se.\texttt{f} \mid se.\texttt{m}(\bar{se}) \mid \texttt{shadow}\ \texttt{C}(t, \bar{se}) \mid \texttt{C.m}(se) \mid \\
 & & Op(\bar{se}) \\
SE & ::= & [\,] \mid SE.\texttt{f} \mid SE.\texttt{m}(\bar{se}) \mid sv.\texttt{m}(\bar{sv}, SE, \bar{se}') \mid \\
 & & \texttt{shadow}\ \texttt{C}(t, \bar{sv}, SE, \bar{se}') \mid \texttt{C.m}(SE) \mid \\
 & & Op(\bar{sv}, SE, \bar{se})
\end{array}$$

The shadowing specification requires that shadow expressions evolve in a shape-conformant way with the original configuration. To this end, we define a metatheoretic function for shadow method bodies, *smbody*, that imposes untainted tags on all method bodies, defined a priori, and removes primitive values.

**Definition 3.6.2** *Shadow method bodies are defined by the function smbody.*

$$smbody_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.srewrite(\texttt{e}),$$

$$
\begin{array}{l}
\mathrm{Shadow}(1, \texttt{shadow TopLevel}(\circ).\texttt{main}(\texttt{shadow C}(\bullet, \bar{\delta}))).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, sv.\texttt{m}(\overline{sv}')) \wedge \texttt{C.m} \notin \mathit{LibMeths} \wedge\\
\quad \mathit{smbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.se' \implies \mathrm{Shadow}(n+1, \mathit{SE}[\texttt{C.m}(se'[\overline{sv}'/\bar{x}][sv/\texttt{this}])]).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, \texttt{shadow C}(t_0, \overline{sv}).\texttt{m}(\overline{\texttt{shadow C}(t, \overline{sv})})) \wedge \texttt{C.m} \in \mathit{LibMeths} \wedge\\
\quad \mathit{smbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{shadow D}(\circ, \overline{se}) \wedge \mathrm{Prop}(t, \texttt{C.m}(t_0, \bar{t})) \implies\\
\quad \mathrm{Shadow}(n+1, \mathit{SE}[\texttt{C.m}(\texttt{shadow D}(t, \overline{se})[\texttt{shadow C}(t_0, \overline{sv})/\texttt{this}][\overline{\texttt{shadow C}(t, \overline{sv})}/\bar{\texttt{x}}])]).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, \texttt{shadow C}(t, \overline{sv}).\texttt{f}_\texttt{i}) \implies \mathrm{Shadow}(n+1, \mathit{SE}[sv_i]).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, \mathit{Op}(\bar{\delta})) \implies \mathrm{Shadow}(n, \mathit{SE}[\delta]).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, \texttt{C.endorse}(\texttt{shadow C}(t, \overline{sv}))) \implies\\
\quad \mathrm{Shadow}(n+1, \mathit{SE}[\texttt{shadow C}(\circ, \overline{sv})]).\\
\mathrm{Shadow}(n, se) \wedge \mathrm{match}(se, \mathit{SE}, \texttt{C.m}(sv)) \wedge \texttt{m} \neq \texttt{endorse} \implies\\
\quad \mathrm{Shadow}(n+1, \mathit{SE}[sv]).
\end{array}
$$

Figure 3.6: Shadow predicate definition.

*where* $\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{e}$ *and the shadow rewriting function,* $\mathit{srewrite}$*, is defined as follows, where* $\mathit{srewrite}(\bar{\texttt{e}})$ *denotes a mapping of* $\mathit{srewrite}$ *over the vector* $\bar{\texttt{e}}$*:*

$$\mathit{srewrite}(\texttt{x}) = \texttt{x}$$

$$\mathit{srewrite}(\texttt{new C}(\bar{\texttt{e}})) = \texttt{shadow C}(\circ, \mathit{srewrite}(\bar{\texttt{e}}))$$

$$\mathit{srewrite}(\texttt{e.f}) = \mathit{srewrite}(\texttt{e}).\texttt{f}$$

$$\mathit{srewrite}(\texttt{e.m}(\bar{\texttt{e}}')) = \mathit{srewrite}(\texttt{e}).\texttt{m}(\mathit{srewrite}(\bar{\texttt{e}}'))$$

$$\mathit{srewrite}(\texttt{C.m}(\texttt{e})) = \texttt{C.m}(\mathit{srewrite}(\texttt{e}))$$

$$\mathit{srewrite}(\mathit{Op}(\bar{\texttt{e}})) = \mathit{Op}(\mathit{srewrite}(\bar{\texttt{e}}))$$

$$\mathit{srewrite}(\nu) = \delta$$

We use $\mathrm{match}$ as a predicate which matches a shadow expression $se$, to a shadow context $\mathit{SE}$ and a shadow expression $se'$ where $se'$ is the part of the shadow in the hole. The definition of $\mathrm{match}$ is given in Figure 3.5.

Next, in Figure 3.6, we define a predicate $\text{Shadow}(n, se)$ where $se$ is the relevant shadow expression at execution step $n$, establishing an ordering for the shadow trace. $\text{Shadow}$ has as its precondition a "current" shadow expression, and as its postcondition the shadow expression for the next step of evaluation (with the exception of the rule for shadowing $Op$s on primitive values which reflects the "immediate" valuation due to the definition of $\approx$– note the timestamp is not incremented in the postcondition in that case). We set the shadow of the initial configuration at timestamp 1, and then $\text{Shadow}$ inductively shadows the full trace. $\text{Shadow}$ is defined by case analysis on the structure of shadow expression in the hole. The shadow expression in the hole and the shadow evaluation context are derived from $\text{match}$ predicate definition.[2] Let's denote shadow of expression $\texttt{e}$ in a given trace $\tau$ by $shadow_\tau(\texttt{e})$.

Note especially how integrity is treated by sanitization and endorsement in this specification. For elements of $Sanitizers$, the result is considered to be endorsed. For library methods, taint is propagated given a user-defined predicate $\text{Prop}(t, T)$ where $T$ is a compound term of the form $\texttt{C.m}(\bar{t})$ with $\bar{t}$ the given integrity of `this` followed by the integrity of the arguments to method `C.m`, and $t$ is the integrity of the result. For example, one could define:

$$\text{meet}(t, t_1, t_2) \Rightarrow \text{Prop}(t, \texttt{String.concat}(t_1, t_2)). \tag{3.1}$$

### 3.6.2 Integrity Taint Analysis Policies

We define our policy for integrity taint analysis. It blocks the execution of the program whenever a tainted value is passed to a secure method. To this end, in Figure 3.7 we define the predicate $\text{BAD}$ which identifies traces that should be rejected as unsafe.

**Definition 3.6.3** *Let $X$ be the set of rules in Figures 3.5, 3.6, and 3.7 and the set of user defined rules for* $\text{Prop}$. *The integrity taint analysis policy is defined as the set of traces that do not end*

[2]Some notational liberties are taken in Figure 3.6 regarding expression and context substitutions, which are defined using predicates elided for brevity.

$$\begin{array}{l} \mathrm{match}(se, SE, \texttt{shadow C}(t, \overline{sv}).\texttt{m}(\texttt{shadow D}(t', \overline{sv}'))) \wedge \mathrm{Shadow}(n, se) \wedge \\ \quad \mathrm{Call}(n, \texttt{C}, \bar{\texttt{v}}, \texttt{m}, \texttt{u}) \wedge \texttt{C.m} \in \mathit{SSOs} \implies \mathrm{SsoTaint}(n, t', \texttt{u}). \\ \mathrm{SsoTaint}(n, \bullet, \texttt{u}) \implies \mathrm{BAD}(n). \end{array}$$

Figure 3.7: Predicates for Specifying Integrity Taint Analysis Policy

*in* BAD *states.*

$$\mathrm{SP}_{\mathrm{taint}} = \{\tau \mid \forall n, \mathit{toFOL}(\tau) \cup X \nvdash \mathrm{BAD}(n)\}.$$

We define a program as being safe iff it does not produce a bad trace.

**Definition 3.6.4** *We call a program* $\texttt{p}(\mathrm{a})$ safe *iff for all* $\tau$ *it is the case that* $\texttt{p}(\mathrm{a}) \Downarrow \tau$ *implies* $\tau \in \mathrm{SP}_{\mathrm{taint}}$. *We call the program* unsafe *iff there exists some trace* $\tau$ *such that* $\texttt{p}(\mathrm{a}) \Downarrow \tau$ *and* $\tau \notin \mathrm{SP}_{\mathrm{taint}}$.

To illustrate the major points of our construction for source program traces and their shadows, we consider an example of program that contains an sso call on a string that has been constructed from a sanitized low integrity input.

**Example 3.6.2** *Let*

$$\begin{array}{l} \mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{new Sec}().\texttt{secureMeth}( \\ \quad \texttt{new Sec}().\texttt{sanitize}(\texttt{x.concat}(\texttt{new String}(''\texttt{world}'')))). \end{array}$$

*Assume the string* ″`hello `″ *is tainted with low integrity. Figure 3.8 gives the source trace and the shadow expressions. For the sake of brevity and clarity in illustrating the main ideas, we have assumed that methods* `Sec.sanitize` *and* `Sec.secureMeth` *are identity functions. Some reduction steps are elided in the example as* $n$*-length multi-step reductions* $\rightarrow^n$.

$$
\begin{array}{l}
\texttt{p(new String("hello "))} \\
\quad \rightarrow^5 \texttt{TopLevel.main(new Sec().secureMeth(} \\
\qquad\qquad \texttt{new Sec().sanitize(new String("hello world"))))} \\
\quad \rightarrow^4 \texttt{TopLevel.main(new Sec().secureMeth(new String("hello world")))} \\
\quad \rightarrow^3 \texttt{new String("hello world")}.
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Shadow}(1, \texttt{shadow TopLevel}(\circ)\texttt{.main(shadow String}(\bullet))) \\
\mathrm{Shadow}(6, \texttt{TopLevel.main(shadow Sec}(\circ)\texttt{.secureMeth(} \\
\qquad\qquad \texttt{shadow Sec}(\circ)\texttt{.sanitize(shadow String}(\bullet))))) \\
\mathrm{Shadow}(10, \texttt{TopLevel.main(shadow Sec}(\circ)\texttt{.secureMeth(shadow String}(\circ)))) \\
\mathrm{Shadow}(13, \texttt{shadow String}(\circ))
\end{array}
$$

Figure 3.8: Example 3.6.2: Source trace and shadow expressions

## 3.7 Sanity Conditions on Library Methods

As noted before, taint is propagated by library methods according to a user-defined predicate $\mathrm{Prop}$. We define two sanity conditions for library methods: *not undertainting* and *not overtainting*. These conditions are required in the implementation in order to establish sound and complete taint analysis.

First we define trust equivalence relation between FJ expressions (states) in order to specify sanity conditions for library methods[3]. We accomplish this by definition of a relation $=_\circ$ on states within a trace $\tau$ that requires identity of high integrity values, but allows low integrity values to be different.

**Definition 3.7.1** *The relation $=_\circ$ on expressions (states) and sequence of expressions within an execution trace $\tau$ is the least relation inductively defined by inference rules in Figure 3.9.*

**Definition 3.7.2** *Let* $\texttt{E}[\texttt{new C}(\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}})] \rightarrow^* \texttt{E}[\texttt{w}]$ *and* $\texttt{E}'[\texttt{new C}(\bar{\texttt{v}}').\texttt{m}(\bar{\texttt{u}}')] \rightarrow^* \texttt{E}'[\texttt{w}']$ *such that* $\texttt{C.m} \in$ *$LibMeths$ and* $\texttt{E}[\texttt{new C}(\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}})]] =_\circ \texttt{E}'[\texttt{new C}(\bar{\texttt{v}}').\texttt{m}(\bar{\texttt{u}}')]$. *We say* $\texttt{C.m}$ is not undertainting *iff*

[3]This definition can also be used for noninterference (Definition 2.2.6) in FJ.

$$\mathtt{x} =_\circ \mathtt{x} \qquad \nu =_\circ \nu \qquad \frac{\mathtt{e}_1 =_\circ \mathtt{e}_2}{\mathtt{e}_1.\mathtt{f} =_\circ \mathtt{e}_2.\mathtt{f}} \qquad \frac{\mathtt{e}_1, \bar{\mathtt{e}}_1 =_\circ \mathtt{e}_2, \bar{\mathtt{e}}_2}{\mathtt{e}_1.\mathtt{m}(\bar{\mathtt{e}}_1) =_\circ \mathtt{e}_2.\mathtt{m}(\bar{\mathtt{e}}_2)} \qquad \frac{\mathtt{e}_1 =_\circ \mathtt{e}_2}{\mathtt{C.m}(\mathtt{e}_1) =_\circ \mathtt{C.m}(\mathtt{e}_2)}$$

$$\frac{\bar{\mathtt{e}}_1 =_\circ \bar{\mathtt{e}}_2 \qquad \mathit{shadow}_\tau(\mathtt{new\ C}(\bar{\mathtt{e}}_1)) = \mathtt{shadow\ C}(\circ, \overline{se_1}) \qquad \mathit{shadow}_\tau(\mathtt{new\ C}(\bar{\mathtt{e}}_2)) = \mathtt{shadow\ C}(\circ, \overline{se_2})}{\mathtt{new\ C}(\bar{\mathtt{e}}_1) =_\circ \mathtt{new\ C}(\bar{\mathtt{e}}_2)}$$

$$\frac{\mathit{shadow}_\tau(\mathtt{new\ C}(\bar{\mathtt{e}}_1)) = \mathtt{shadow\ C}(\bullet, \overline{se_1}) \qquad \mathit{shadow}_\tau(\mathtt{new\ D}(\bar{\mathtt{e}}_2)) = \mathtt{shadow\ D}(\bullet, \overline{se_2})}{\mathtt{new\ C}(\bar{\mathtt{e}}_1) =_\circ \mathtt{new\ D}(\bar{\mathtt{e}}_2)}$$

$$\frac{\mathtt{e}_1^1 =_\circ \mathtt{e}_1^2 \ \cdots \ \mathtt{e}_n^1 =_\circ \mathtt{e}_n^2}{\mathtt{e}_1^1 \cdots \mathtt{e}_n^1 =_\circ \mathtt{e}_1^2 \cdots \mathtt{e}_n^2}$$

Figure 3.9: Definition of $=_\circ$ relation.

$\mathtt{E}[\mathtt{w}] =_\circ \mathtt{E}'[\mathtt{w}']$.

**Definition 3.7.3** *Let* `C.m` *be a library methods that is constant. We say* `C.m` is not overtainting *iff* $\mathrm{Prop}(\circ, \mathtt{C.m}(t_1, \overline{t_2}))$.

For example, `String.concat` is not undertainting if the taint propagation policy is defined as (3.1), but if it is defined as $\mathrm{Prop}(\circ, \mathtt{String.concat}(t_1, t_2))$ then this library method is undertainting. As an example for overtainting library methods, consider primitive operation $\mathit{prim}(\mathit{str})$ to be a constant function over strings with return value 0. Let OO wrapper for $\mathit{prim}$ be library method `String.primwrapper` such that

$$\mathit{mbody}_{CT}(\mathtt{primwrapper}, \mathtt{String}) = \varnothing, \mathtt{new\ Int}(\mathit{prim}(\mathtt{this.val}))$$

with propagation policy $\mathrm{Prop}(t, \mathtt{String.primwrapper}(t))$.

## 3.8 Examples

In what follows, we give examples to show incomparability of noninterference and explicit integrity in FJ. In this regard, Example 3.8.1 demonstrates a program which satisfies explicit

$$\texttt{new TopLevel().main(new C("hello"))} \rightarrow^*_{f_1}$$
$$\texttt{TopLevel.main(C.m(new C("hello").sso(new C("hi"))))} \rightarrow^*_{f_2} \texttt{new C("hi")}$$

$$f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\texttt{TopLevel.main(C.m(a.sso(new C("hi"))))}, \epsilon)$$
$$f_2(\mathrm{a}, \texttt{TopLevel.main(C.m(u.sso(v)))}) = (\texttt{v}, ie(\texttt{v}))$$
$$f_2 * f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\texttt{new C("hi")}, ie(\texttt{new C("hi")}))$$

$$p_e(\mathrm{a}, \texttt{new TopLevel().main(u)}, f_1) = p_e(\mathrm{a}, \texttt{new TopLevel().main(u)}, f_2 * f_1) =$$
$$\{\mathrm{a}' \mid \mathrm{a}' = \texttt{new C}(str) \vee \mathrm{a}' = \texttt{new D}(str)\}$$

$$\texttt{new TopLevel().main(new C("hello"))} =_{\circ} \texttt{new TopLevel().main(new D("hello"))}$$

Figure 3.10: Example 3.8.1: The execution trace, state transformers, attacker powers and trust equivalent initial states.

integrity while it is interfering. Conversely, the program given in Example 3.8.2 satisfies noninterference but not explicit integrity. Example 3.8.3 discusses a case where the program satisfies explicit integrity modulo endorsement. In the following examples, method `C.sso` $\in SSOs$ is assumed to be an identity function. Moreover, the shadow of $\mathrm{a}$ is denoted $sa$.

**Example 3.8.1** *Let* $mbody_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$. *Also assume that* $\texttt{C} <: \texttt{D}$, *and we have* $mbody_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{x.sso(new D("hi"))}$ *and* $mbody_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{x.sso(new C("hi"))}$. *Whether the user input to* `TopLevel.main` *is an object of class* `C` *or* `D`*, different methods* `m` *are invoked by dynamic dispatch. Let the attack be* `new C("hello")`*. Corresponding execution trace, state transformer definitions for any* $\mathrm{a}$*,* `u` *and* `v`*, and attacker powers are given in Figure 3.10. Then, according to the composition of state transformers, the attacker power is preserved. Thus, explicit integrity is satisfied. However, this program does not satisfy noninterference. Considering trust equivalent initial states depicted in Figure 3.10, the attacks* `new C("hello")` *and* `new D("hello")` *generate unequal sequence of events* $[ie(\texttt{new C("hi")})]$ *and* $[ie(\texttt{new D("hi")})]$.

$$\texttt{new TopLevel().main(new C())} \rightarrow^*_{f_1} \texttt{TopLevel.main(C.m(new C().sso(new C())))} \rightarrow^*_{f_2} \texttt{new C()}$$

$$f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\texttt{TopLevel.main(C.m(a.sso(a)))}, \epsilon)$$
$$f_2(\mathrm{a}, \texttt{TopLevel.main(C.m(u.sso(u)))}) = (\mathrm{u}, ie(\mathrm{u}))$$
$$f_2 * f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\mathrm{a}, ie(\mathrm{a}))$$

$$p_e(\mathrm{a}, \texttt{new TopLevel().main(u)}, f_2 * f_1) = \{\mathrm{a}\}$$

$$\texttt{new TopLevel().main(new C())} =_\circ \texttt{new TopLevel().main(new D())}$$

Figure 3.11: Example 3.8.2: The execution trace, state transformers, attacker powers and trust equivalent initial states.

**Example 3.8.2** *Let* $mbody_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$*. Also assume that* $\texttt{C} <: \texttt{D}$*, and we have* $mbody_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{x.sso(new C())}$ *and* $mbody_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{x.sso(x)}$*. Note that we assume no fields for classes* C *and* D*, and so only objects* `new C()` *and* `new D()` *are definable. Let the attack be* `new C()`*. Similar to Example 3.8.1, details are given in Figure 3.11. Since the explicit attacker power is refined to the attack, explicit integrity is not satisfied. However, this program satisfies noninterference. The attacks* `new C()` *and* `new D()` *both generate the same event* $ie(\texttt{new C()})$ *considering trust equivalent initial states given in Figure 3.11.*

**Example 3.8.3** *Let* $mbody_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$*. Also assume that* $\texttt{C} <: \texttt{D}$*, and we have* $mbody_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{x.sso(new D("hi"))}$ *and* $mbody_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{x.sso(x.endorse())}$*. Let the attack be* `new C("hello")`*. Details are given in Figure 3.12. As explicit attacker power is not further refined after endorsement, explicit integrity modulo endorsement is satisfied. However, the program is not satisfying noninterference, since the attacks* `new C("hello")` *and* `new D("hello")` *generate different sequences of events.*

$$\texttt{new TopLevel().main(new C("hello"))} \rightarrow^*_{f_1}$$
$$\texttt{TopLevel.main(C.m(new C("hello").sso(new C("hello"))))} \rightarrow^*_{f_2} \texttt{new C("hello")}$$

$$f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\texttt{TopLevel.main(C.m(}\mathrm{a}\texttt{.sso(}\mathrm{a}\texttt{)))}, en(\mathrm{a}))$$
$$f_2(\mathrm{a}, \texttt{TopLevel.main(C.m(v.sso(v))))} = (\texttt{v}, ie(\texttt{v}))$$
$$f_2 * f_1(\mathrm{a}, \texttt{new TopLevel().main(u)}) = (\mathrm{a}, [en(\mathrm{a}), ie(\mathrm{a})])$$

$$p_e(\mathrm{a}, \texttt{new TopLevel().main(u)}, f_1) = p_e(\mathrm{a}, \texttt{new TopLevel().main(u)}, f_2 * f_1) = \{\mathrm{a}\}$$

$$\texttt{new TopLevel().main(new C("hello"))} =_\circ \texttt{new TopLevel().main(new D("hello"))}$$

Figure 3.12: Example 3.8.3: The execution trace, state transformers, attacker powers and trust equivalent initial states.

# Chapter 4

# Taint Analysis Instrumentation via Program Rewriting

Now we define an object based dynamic integrity taint analysis in a more familiar operational style. Taint analysis instrumentation is added automatically by a program rewriting algorithm $\mathcal{R}$ that models Phosphor rewriting algorithm. It adds taint label fields to all objects, and operations for appropriately propagating taint along direct flow paths. We incorporate blocking behavior to enforce blocking checks at secure sinks. In the next Chapter we will show that this analysis is sound and complete with respect to explicit integrity modulo endorsement, assuming library methods are neither over nor undertainting.

## 4.1 Taint Analysis Instrumentation

The target language of the rewriting algorithm $\mathcal{R}$, called $\mathrm{FJ}_{\mathrm{taint}}$, is the same as FJ, except we add taint labels $t$ as a form of primitive value $\nu$, the type of which we posit as `Taint`. For the semantics of taint values operations we define:

$$\vee(t_1, t_2) \approx t_1 \vee t_2 \qquad\qquad \wedge(t_1, t_2) \approx t_1 \wedge t_2$$

In addition we introduce a "check" operation $?$ such that $?t \approx t$ iff $t > \bullet$. We also add an explicit sequencing operation of the form $\mathtt{e};\mathtt{e}$ to target language expressions, and evaluation contexts of the form $\mathtt{E};\mathtt{e}$. along with the appropriate operational semantics rule that we define below in Section 4.3.

Now we define the program rewriting algorithm $\mathcal{R}$ as follows. Since in our security model the only tainted input source is a specified argument to a top-level program, the rewriting algorithm adds an untainted label to all objects. The class table is then manipulated to specify a `taint` field for all objects and a `check` object method that blocks if the argument is tainted.

**Definition 4.1.1** *For any expression* $\mathtt{e}$*, the expression* $\mu(\mathtt{e})$ *is syntactically equivalent to* $\mathtt{e}$ *except with every subexpression* $\mathtt{new\ C}(\bar{\mathtt{e}})$ *replaced with* $\mathtt{new\ C}(\circ, \bar{\mathtt{e}})$*. Given SSOs, define* $\mathcal{R}(\mathtt{e}, CT) = (\mu(\mathtt{e}), \mathcal{R}(CT))$*, where* $\mathcal{R}(CT)$ *is the smallest class table satisfying the axioms given in Figure 4.1.*

## 4.2 Taint Propagation of Library Methods

Another important element of taint analysis is instrumentation of library methods that propagate taint– the propagation must be made explicit to reflect the interference of arguments with results. The approach to this in taint analysis systems is often motivated by efficiency as much as correctness [18]. We assume that library methods are instrumented to propagate taint as intended (i.e. in accordance with the user defined predicate $\mathrm{Prop}$).

Here is how string concatenation, for example, can be modified to propagate taint according to the policy given by (3.1) in Section 3.6.1. Note the taint of arguments will be propagated to

$$\textit{fields}_{\mathcal{R}(CT)}(\texttt{Object}) = \texttt{Taint taint} \qquad \textit{mbody}_{\mathcal{R}(CT)}(\texttt{check}, \texttt{Object}) = \texttt{x}, \texttt{new Object(?x.taint)}$$

$$\frac{\texttt{C.m} \in \textit{SSOs} \qquad \textit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{e}}{\textit{mbody}_{\mathcal{R}(CT)}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{this.check(x)}; \mu(\texttt{e})} \qquad \frac{\texttt{C.m} \notin \textit{SSOs} \qquad \textit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}}{\textit{mbody}_{\mathcal{R}(CT)}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \mu(\texttt{e})}$$

Figure 4.1: Axioms for Rewriting Algorithm

results by taking the meet of argument taint, thus reflecting the degree of integrity corruption:

```
String concat(String x)
        { return(new String(∧(this.taint, x.taint), @(this.val, x.val))); }
```

## 4.3 Operational Semantics of FJ$_{\mathrm{taint}}$

We “inherit” the reduction semantics of FJ, and add a rule also for evaluation of sequencing.

Sequence

$$\texttt{v}; \texttt{e} \rightarrow \texttt{e}$$

As for FJ we use $\rightarrow^*$ to denote the reflexive, transitive closure on $\rightarrow$ over FJ$_{\mathrm{taint}}$ configurations of the form $\texttt{e}$. We define FJ$_{\mathrm{taint}}$ configurations and traces as for FJ. Abusing notation, we write $\mathcal{R}(\mathfrak{p}(\mathrm{a})) \Downarrow \tau$ iff $\tau$ begins with the configuration $\mathcal{R}(\mathfrak{p}(\mathrm{a}))$, and also $\kappa \Downarrow \tau$ iff $\tau$ is a valid trace in the FJ$_{\mathrm{taint}}$ semantics beginning with $\kappa$.

# Chapter 5

# Soundness and Completeness of Program Rewriting

In this Chapter we show how explicit integrity semantics (introduced in Chapter 2 and instantiated in Section 3.5) is used to establish soundness and completeness conditions for $\mathcal{R}$. In order to specify soundness and completeness for $\mathcal{R}$, we define the notion of *security failure*.

**Definition 5.0.1** *An FJ*$_{\text{taint}}$ *program* $\mathtt{e}$ causes a security failure *iff for some* $\mathtt{E}$*,* $\mathtt{v}$*, and* $\mathtt{new\ C}(\bullet, \bar{\mathtt{v}})$*, we have* $\mathtt{e} \rightarrow^* \mathtt{E}[\mathtt{v.check}(\mathtt{new\ C}(\bullet, \bar{\mathtt{v}}))]$.

This notion can be used to specify the soundness and completeness of $\mathcal{R}$ with respect to the explicit integrity semantics.

**Definition 5.0.2** *Rewriting algorithm* $\mathcal{R}$ *is* sound wrt explicit integrity semantics *iff for any FJ program* $\mathtt{p}$ *and any attack* $\mathrm{a}$*,* $\mathtt{p}(\mathrm{a})$ *satisfies explicit integrity modulo endorsement provided that program* $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ *does not cause a security failure.*

**Definition 5.0.3** *Rewriting algorithm $\mathcal{R}$ is* complete wrt explicit integrity semantics *iff for any FJ program $\mathfrak{p}$ and any attack $\mathrm{a}$, program $\mathcal{R}(\mathfrak{p}(\mathrm{a}))$ does not cause a security failure provided that $\mathfrak{p}(\mathrm{a})$ satisfies explicit integrity modulo endorsement.*

Establishing soundness and completeness of $\mathcal{R}$ require it to be a *faithful* implementation of integrity taint policy. We call $\mathcal{R}$ faithful provided that for any FJ program $\mathfrak{p}$ and attack $\mathrm{a}$, program $\mathfrak{p}(\mathrm{a})$ is unsafe (Definition 3.6.4) iff $\mathcal{R}(\mathfrak{p}(\mathrm{a}))$ causes a security failure (Definition 5.0.1). The detailed proofs of faithfulness of $\mathcal{R}$ and subsequently its soundness and completeness are given below.

## 5.1 Semantics Preservation

A core condition for correctness of $\mathcal{R}$ is proof of semantics preservation for safe programs in FJ, i.e. that rewritten programs simulate the semantics of source program modulo security instrumentations. The way this simulation is defined will naturally imply a full and faithful implementation of taint shadowing semantics. Adapting the Definition in [23], we say that rewriting algorithm $\mathcal{R}$ is semantics preserving for $\mathrm{SP}_{\mathrm{taint}}$ iff there exists a relation $:\approx$ with the following property.

**Definition 5.1.1** *Rewriting algorithm $\mathcal{R}$ is* semantics preserving *iff for all safe programs $\mathfrak{p}(\mathrm{a})$ (Definition 3.6.4) all of the following hold:*

1. *For all traces $\tau$ such that $\mathfrak{p}(\mathrm{a}) \Downarrow \tau$ there exists $\tau'$ with $\tau :\approx \tau'$ and $\mathcal{R}(\mathfrak{p}(\mathrm{a})) \Downarrow \tau'$.*

2. *For all traces $\tau$ such that $\mathcal{R}(\mathfrak{p}(\mathrm{a})) \Downarrow \tau$ there exists a trace $\tau'$ such that $\tau' :\approx \tau$ and $\mathfrak{p}(\mathrm{a}) \Downarrow \tau'$.*

Observe that $:\approx$ may relate more than one trace in the target program to a trace in the source program, since instrumentation in the target language may introduce new reduction steps that can cause "stuttering" with respect to source language traces.

$$
\mathit{overlay}(\mathtt{x},\mathtt{x}) = \mathtt{x} \qquad \mathit{overlay}(\nu,\delta) = \nu \qquad \mathit{overlay}(\mathit{Op}(\bar{\mathtt{e}}),\, \mathit{Op}(\overline{se})) = \mathit{Op}(\overline{\mathit{overlay}(\mathtt{e}, se)})
$$

$$
\mathit{overlay}(\mathtt{e.f}, se.\mathtt{f}) = \mathit{overlay}(\mathtt{e}, se).\mathtt{f} \qquad \mathit{overlay}(\mathtt{new\ C}(\bar{\mathtt{e}}), \mathtt{shadow\ C}(t, \overline{se})) = \mathtt{new\ C}(t, \overline{\mathit{overlay}(\mathtt{e}, se)})
$$

$$
\mathit{overlay}(\mathtt{C.m(e)}, \mathtt{C.m}(se)) = \mathtt{C.m}(\mathit{overlay}(\mathtt{e}, se)) \qquad \mathit{overlay}(\mathtt{e.m}(\bar{\mathtt{e}'}), se.\mathtt{m}(\overline{se'})) = \mathit{overlay}(\mathtt{e}, se).\mathtt{m}(\overline{\mathit{overlay}(\mathtt{e}', se')})
$$

Figure 5.1: Definition of *overlay*.

As evidenced in the statement of semantics preservation, we will generally relate "executable" source programs $\mathtt{p}(\mathrm{a})$ with rewritten programs $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ for simplicity in the statement of properties and ease of proofs. However, for practical purposes it is important to observe that instrumentation can be performed on program entry points $\mathtt{p}$ and class tables $CT$ once, prior to invocation on possibly tainted $\mathrm{a}$, due to the following property which follows immediately from the definition of $\mathcal{R}$.

**Lemma 5.1.1** $\mathcal{R}(\mathtt{p}(\mathrm{a})) = \mathcal{R}(\mathtt{p})(\mathcal{R}(\mathrm{a}))$

To establish correctness of program rewriting, we need to define a correspondence relation $:\approx$. Source language execution traces and target language execution traces correspond if they represent the same expression evaluated to the same point. We make a special case: when a sink method is called in the source execution, in which the target execution needs to first check the arguments to the sink method in order to enforce taint policy by `check`. In this case, the target execution may be ahead by a step, allowing time to enforce taint policy.

In order to define the correspondence between execution traces of the source and target language, we first define a mapping, *overlay*, that computes the target configuration by overlaying the source configuration with its shadow.

**Definition 5.1.2** *The mapping* $\mathit{overlay} : (\mathtt{e}, se) \mapsto \mathtt{e}'$ *is defined in Figure 5.1.*

We define a way to obtain the last shadow in a trace. Give a source trace $\tau$ of length $n$, $\mathit{LastShadow}(\tau)$ denotes the shadow of the last configuration in the trace $\tau$. Considering the

$$trim(\mathtt{x}) = \mathtt{x} \qquad trim(\mathtt{e.f}) = trim(\mathtt{e}).\mathtt{f} \qquad trim(\mathtt{new\ C}(\bar{\mathtt{e}})) = \mathtt{new\ C}(\overline{trim(\mathtt{e})}) \qquad trim(\mathtt{C.m}(\mathtt{e})) = \mathtt{C.m}(trim(\mathtt{e}))$$

$$trim(Op(\bar{e})) = Op(\overline{trim(e)}) \qquad trim(\mathtt{e}_1;\mathtt{e}_2) = trim(\mathtt{e}_1); trim(\mathtt{e}_2)$$

$$trim(\mathtt{e.m}(\bar{\mathtt{e}}')) = \begin{cases} \epsilon & \text{if } \mathtt{m} = \mathtt{check} \\ trim(\mathtt{e}).\mathtt{m}(\overline{trim(\mathtt{e}')}) & \text{if } \mathtt{m} \neq \mathtt{check} \end{cases}$$

Figure 5.2: Definition of $trim$.

rule

$$\text{Shadow}(n, se) \implies \text{LShadow}(se), \tag{5.1}$$

we define $LastShadow(\tau) = se$ such that $\lfloor\tau\rfloor \otimes X \vdash \text{LShadow}(se)$, where $X$ contains the rules given in Figure 3.5, Figure 3.6 and (5.1). We need to show that $LastShadow$ is total function on non-trivial traces, i.e., $LastShadow$ uniquely maps any non-empty trace to a shadow expression.

**Lemma 5.1.2** *$LastShadow$ is total function on non-trivial traces.*

*Proof.* By induction on the length of traces and the fact that shadow expressions are defined uniquely for every step of reduction in Figure 3.6. □

We also define a mapping, $trim$, from the expressions of the target language to the expressions of the source language. Intuitively, $trim$ removes the invocations to `check`.

**Definition 5.1.3** *The mapping $trim : \mathtt{e} \mapsto \mathtt{e}'$ is defined in Figure 5.2. We assume $\epsilon$ to be no-op, i.e., $\epsilon; \mathtt{e} = \mathtt{e}$.*

**Definition 5.1.4** *Given source language execution trace $\tau = \sigma\kappa$ and target language execution trace $\tau' = \sigma'\kappa'$, $\tau :\approx \tau'$ iff $overlay(\kappa, LastShadow(\tau)) = trim(\kappa')$.*

In what follows, we prove the semantics preservation given by Definition 5.1.1. To this end, in Lemma 5.1.3, we show that if $\mathit{trim}$ of an expression is a value, that expression eventually reduces to that value provided it is not a security failure. Moreover, Lemma 5.1.4 states that if $\mathit{trim}$ of a non-security failure expression $\mathtt{e}$ is reduced to $\mathtt{e}'$ then $\mathtt{e}$ reduces (in potentially multiple steps) to some expression with the same $\mathit{trim}$ as $\mathtt{e}'$.

**Lemma 5.1.3** *For all $FJ_{\text{taint}}$ expressions $\mathtt{e}$, if $\mathit{trim}(\mathtt{e}) = \mathtt{v}$, then either (1) there exists some trace $\sigma$ such that $\mathtt{e} \Downarrow \sigma\mathtt{v}$, or (2) $\mathtt{e}$ causes a security failure.*

*Proof.* By induction on the structure of $\mathtt{e}$. □

**Lemma 5.1.4** *For all $FJ_{\text{taint}}$ expressions $\mathtt{e}$, if $\mathit{trim}(\mathtt{e}) \rightarrow \mathtt{e}'$ then either (1) there exists $\sigma$ such that $\mathtt{e} \Downarrow \sigma\mathtt{e}''$ with $\mathit{trim}(\mathtt{e}'') = \mathit{trim}(\mathtt{e}')$, or (2) $\mathtt{e}$ causes a security failure.*

*Proof.* By induction on the structure of $\mathtt{e}$, and applying Lemma 5.1.3. □

Lemma 5.1.5 states that $\mathit{overlay}$ing a method body with its shadow is equal to the same method body in the rewritten class table.

**Lemma 5.1.5** $\mathit{overlay}(\mathtt{e}, \mathit{srewrite}(\mathtt{e})) = \mu(\mathtt{e})$.

*Proof.* By induction on the structure of $\mathtt{e}$. □

Lemma 5.1.6 and Lemma 5.1.7 state that single step and multi step reductions in FJ preserve $:\approx$.

**Lemma 5.1.6** *If $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ and $\mathtt{e}_1 \rightarrow \mathtt{e}_1'$ then there exists $\sigma$ such that $\kappa_2 \Downarrow \sigma$ and $\tau_1\mathtt{e}_1\mathtt{e}_1' :\approx \tau_2\sigma$.*

*Proof.* By induction on the derivation of $\mathtt{e}_1 \rightarrow \mathtt{e}_1'$ and applying Lemmas 5.1.4 and 5.1.5. □

**Lemma 5.1.7** *If* $\tau_1 \mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathtt{e}_1 \Downarrow \sigma_1$*, then there exists* $\sigma_2$ *such that* $\kappa_2 \Downarrow \sigma_2$ *and* $\tau_1\sigma_1 :\approx \tau_2\sigma_2$.

*Proof.* By induction on the derivation of $\mathtt{e}_1 \Downarrow \sigma_1$ and applying Lemma 5.1.6. □

Similarly, Lemma 5.1.8 and Lemma 5.1.9 argue that single step and multi step reductions in $\text{FJ}_{\text{taint}}$ preserve $:\approx$.

**Lemma 5.1.8** *If* $\tau_1 \mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\kappa_2 \rightarrow \kappa_2'$ *then there exists* $\sigma$ *where* $\mathtt{e}_1 \Downarrow \sigma$ *and* $\tau_1\sigma :\approx \tau_1\kappa_2\kappa_2'$.

*Proof.* By induction on the derivation of $\kappa_2 \rightarrow \kappa_2'$ and applying Lemma 5.1.5. □

**Lemma 5.1.9** *If* $\tau_1 \mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\kappa_2 \Downarrow \sigma_2$*, then there exists* $\sigma_1$ *such that* $\mathtt{e}_1 \Downarrow \sigma_1$ *and* $\tau_1\sigma_1 :\approx \tau_2\sigma_2$.

*Proof.* By induction on the derivation of $\kappa_2 \Downarrow \sigma_2$ and applying Lemma 5.1.8. □

Lemma 5.1.10 states that initial configuration in $\text{FJ}_{\text{taint}}$ corresponds to the initial configuration in FJ. Finally, in Theorem 5.1.1, the semantics preservation property is proven.

**Lemma 5.1.10** *Let* $\mathrm{a} = \mathtt{new}\ \mathtt{C}(\bar{v})$*. Then,*

$$\mathtt{new}\ \mathtt{TopLevel}().\mathtt{main}(\mathrm{a}) :\approx \mathtt{new}\ \mathtt{TopLevel}(\circ).\mathtt{main}(\mathtt{new}\ \mathtt{C}(\bullet, \bar{v})).$$

*Proof.* By the definition of shadow expressions and $:\approx$. □

Theorem 5.1.1 establishes semantics preservation for rewriting algorithm $\mathcal{R}$.

**Theorem 5.1.1** *The rewriting algorithm* $\mathcal{R}$ *is semantics preserving (Definition 5.1.1).*

*Proof.* Lemma 5.1.10 states that initial configuration of a program $\mathfrak{p}$ corresponds to the initial configuration of $\mathcal{R}(\mathfrak{p})$. Lemmas 5.1.7 and 5.1.9 extend the correspondence relation for traces of arbitrary lengths. More specifically, Lemmas 5.1.10 and 5.1.7 entail the 1st condition of Definition 5.1.1, and Lemmas 5.1.10 and 5.1.9 result in the 2nd condition of Definition 5.1.1.

$\square$

## 5.2 Faithfulness of Rewriting Algorithm to Taint Policy

Proof of semantics preservation establishes faithfulness of $\mathcal{R}$ to taint policy, since $\mathrm{SP}_{\mathrm{taint}}$ expresses the taint policy specification as a safety property.

**Definition 5.2.1** *We call rewriting algorithm $\mathcal{R}$ faithful provided that a program $\mathfrak{p}(\mathrm{a})$ is unsafe (Definition 3.6.4) iff $\mathcal{R}(\mathfrak{p}(\mathrm{a}))$ causes a security failure (Definition 5.0.1).*

In order to prove that $\mathcal{R}$ is faithful (Definition 5.2.1), we first need to show that $\mathrm{SP}_{\mathrm{taint}}$ is a safety property.

**Lemma 5.2.1** $\mathrm{SP}_{\mathrm{taint}}$ *is a safety property.*

*Proof.* Let $\tau \notin \mathrm{SP}_{\mathrm{taint}}$. Then there exists some $n$ such that $\mathit{toFOL}(\tau) \cup X \vdash \mathrm{BAD}(n)$. Let $\tau[\cdots n]$ denote the finite prefix of $\tau$ up to timestamp $n$. By Definition 3.6.3 BAD only refers to events that precede step $n$, so it follows that $\mathit{toFOL}(\tau) \cup X \vdash \mathrm{BAD}(n)$ iff $\mathit{toFOL}(\tau[\cdots n]) \cup X \vdash \mathrm{BAD}(n)$, i.e. $\tau \notin \mathrm{SP}_{\mathrm{taint}}$ iff $\tau[\cdots n] \notin \mathrm{SP}_{\mathrm{taint}}$ for finite $n$, hence $\mathrm{SP}_{\mathrm{taint}}$ is a safety property [24]. $\square$

Theorem 5.2.1 establishes faithfulness of $\mathcal{R}$ to taint policy.

**Theorem 5.2.1** *The rewriting algorithm $\mathcal{R}$ is faithful (Definition 5.2.1).*

*Proof.* Suppose on the one hand that $\mathtt{p}(\mathrm{a})$ is unsafe, which is the case iff $\mathtt{p}(\mathrm{a}) \Downarrow \tau$ and $\tau \notin \mathrm{SP}_{\mathrm{taint}}$ for some $\tau$. Then according to Lemma 5.2.1, there exists some timestamp $n$ such that $\tau[\cdots n]$ characterizes $\mathrm{SP}_{\mathrm{taint}}$, i.e., $\mathrm{BAD}(n)$ is derivable from the rules in Definition 3.6.3 and so $(\tau[\cdots n])\sigma \notin \mathrm{SP}_{\mathrm{taint}}$ for any $\sigma$. Let $n$ be the least timestamp with such property and $\tau' = \tau[\cdots n-1]$, and $\tau[n] = \mathtt{E}[\mathtt{v.m}(\mathtt{new\,D}(\bar{\mathtt{u}}))]$ where $\mathtt{v.m}$ is an sso invocation and $\mathtt{new\,D}(\bar{\mathtt{u}})$ has low integrity by the Definition 3.6.3. By Theorem 5.1.1 there exists some trace $\sigma$, such that $\mathcal{R}(\mathtt{p}(\mathrm{a})) \Downarrow \sigma$ and $\tau' :\approx \sigma$. Therefore, by definition of $:\approx$ and $\rightarrow$ we may assert that $\mathit{tail}(\sigma) \Downarrow \kappa_1\kappa_2$ such that

$$\kappa_1 = \mathtt{E}'[\mathtt{v.m}(\mathtt{new\,D}(\bullet, \bar{\mathtt{u}}))]$$

$$\kappa_2 = \mathtt{E}'[\mathtt{v.check}(\mathtt{new\,D}(\bullet, \bar{\mathtt{u}})); \mathtt{e}]$$

Thus, $\mathcal{R}(\mathtt{p}(\mathrm{a})) \Downarrow \sigma\kappa_1\kappa_2$ and therefore $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ causes a security failure.

Supposing on the other hand that $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ causes a security failure, it follows that $\mathtt{p}(\mathrm{a})$ is unsafe by similar reasoning (i.e. fundamental appeal to Theorem 5.1.1), since security `checks` are only added to the beginning of *SSOs* and fail if the argument has low integrity. □

## 5.3 Soundness and Completeness of Rewriting Algorithm

Theorem 5.3.1 and Theorem 5.3.2 establish soundness and completeness of $\mathcal{R}$ respectively. Let the state transformer defined over a trace $\tau$ in FJ be denoted by $st(\tau, f)$.

**Definition 5.3.1** *We define the induced state transformer for an* $FJ_{\mathrm{taint}}$ *trace* $\tau'$ *as* $ist(\tau', f)$, *where* $\tau :\approx \tau'$, $st(\tau, f)$ *and* $head(\tau')$ *does not cause a security failure.*

**Lemma 5.3.1** *Let* $\mathtt{e}_1 \rightarrow \mathtt{e}'_1$, $\mathtt{e}_2 \rightarrow \mathtt{e}'_2$, $\mathtt{e}_1 =_\circ \mathtt{e}'_1$ *and* $\mathtt{e}$ *is not a library method invocation. Then* $\mathtt{e}_2 =_\circ \mathtt{e}'_2$.

*Proof.* By induction on $\mathtt{e} \to \mathtt{e}'$. □

**Lemma 5.3.2** *Let* $\mathtt{e}_1 \to^* \mathtt{e}_1'$, $\mathtt{e}_2 \to^* \mathtt{e}_2'$, $\mathtt{e}_1 =_\circ \mathtt{e}_1'$ *and none of library methods be undertainting. Then* $\mathtt{e}_2 =_\circ \mathtt{e}_2'$.

*Proof.* By induction on $\mathtt{e} \to^* \mathtt{e}'$ using Lemma 5.3.1. □

**Theorem 5.3.1** *The rewriting algorithm $\mathcal{R}$ is sound with respect to explicit integrity semantics (Definition 5.0.2) provided that there does not exist any undertainting library method (Definition 3.7.2).*

*Proof.* If $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ does not cause a security failure, then $\mathtt{p}(\mathrm{a})$ is a safe program according to faithfulness of $\mathcal{R}$ (Theorem 5.2.1). Then, for any trace $\tau'$ such that $\mathcal{R}(\mathtt{p}(\mathrm{a})) \Downarrow \tau'$ there exists a trace $\sigma\sigma'$ such that $\mathtt{p}(\mathrm{a}) \Downarrow \sigma\sigma'$ and $\sigma\sigma' :\approx \tau'$ (Theorem 5.1.1). Let $st(\sigma, f)$ and $st(\mathit{tail}(\sigma)\sigma', g)$. Then, according to the definition of state transformers, the state transformer defined over $\sigma\sigma'$ is $g * f$, i.e., $st(\sigma\sigma', g * f)$. Moreover, based on Definition 5.3.1, $ist(\tau', g * f)$.

In order to prove that $\mathtt{p}(\mathrm{a})$ satisfies explicit integrity modulo endorsement, we need to show that if $\pi_2(g(\mathrm{a}', \mathit{tail}(\sigma))) \notin \mathbf{E_{en}}^*$ for any attack $\mathrm{a}'$, then $\pi_2(f(\mathrm{a}, \mathtt{e}_0)) = \pi_2(f(\mathrm{a}'', \mathtt{e}_0))$ implies that $\pi_2((g{*}f)(\mathrm{a}, \mathtt{e}_0)) = \pi_2((g{*}f)(\mathrm{a}'', \mathtt{e}_0))$ for arbitrary $\mathrm{a}''$ where $\mathtt{e}_0 = \mathtt{new\ TopLevel().main(u)}$ for some $\mathtt{u}$. This is accomplished by induction on the length of $\sigma'$.

When $\sigma'$ has length 0, $\pi_2(g(\mathrm{a}', \mathit{tail}(\sigma))) = \epsilon$ and so the result is immediate. For the inductive phase, we show that the property holds for $\sigma'$ of length $n + 1$, assuming that for $n$-length $\sigma'$ the property is met. So, let's assume the state transformer corresponding to $n$-length $\sigma'$ is $g_1$ and the $(n + 1)$'s step defines state transformer $g_2$. By induction hypothesis we have $\pi_2((g_1 * f)(\mathrm{a}, \mathtt{e}_0)) = \pi_2((g_1 * f)(\mathrm{a}'', \mathtt{e}_0))$ for arbitrary $\mathrm{a}''$. The goal is to show that $\pi_2((g_2 * g_1 * f)(\mathrm{a}, \mathtt{e}_0)) = \pi_2((g_2 * g_1 * f)(\mathrm{a}'', \mathtt{e}_0))$ for arbitrary $\mathrm{a}''$. The only interesting case for the last step of execution is where a sso is invoked (In the rest of the cases integrity events are not generated and so the result is immediately). So, let the $n$th configuration of $\sigma'$ be $\mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\mathtt{new\ D}(\bar{\mathtt{u}}))]$ where $\mathtt{C.m} \in \mathit{SSOs}$. Then considering $\mathit{mbody}_{CT}(\mathtt{m},\mathtt{C}) = \mathtt{x},\mathtt{e}$, $g_2$ is defined as

$$g_2(\mathrm{a}'', \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\mathtt{new\ D}(\bar{\mathtt{u}}))]) =$$
$$(\mathtt{E}[\mathtt{e}[\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}][\mathtt{new\ D}(\bar{\mathtt{u}})/\mathtt{x}]], \mathit{ie}(\mathtt{new\ D}(\bar{\mathtt{u}}))).$$

Note that $\mathtt{new\ D}(\bar{\mathtt{u}})$ is not tainted in its shadow, since $\mathtt{p}(\mathrm{a})$ is a safe program. By Lemma 5.3.2, since $\mathtt{p}(\mathrm{a}) =_\circ \mathtt{p}(\mathrm{a}'')$, the invocations to C.m are trust equivalent. Since ssos are only applied on primitive objects, if two such untainted objects are trust equivalent, then they are syntactically equal. Thus non-tainted object being fed to the sso C.m under attacks a and $\mathrm{a}''$ is of the form $\mathtt{new\ D}(\bar{\mathtt{u}})$. This completes the proof. □

**Theorem 5.3.2** *The rewriting algorithm $\mathcal{R}$ is complete with respect to explicit integrity semantics (Definition 5.0.3) provided that there does not exist any overtainting library method (Definition 3.7.3).*

*Proof.* According to Definition 5.0.3, $\mathcal{R}$ is complete wrt to explicit integrity semantics iff for any FJ program p and any attack a, program $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ does not cause a security failure provided that $\mathtt{p}(\mathrm{a})$ satisfies explicit integrity modulo endorsement. Equivalently, $\mathcal{R}$ is complete wrt to explicit integrity semantics iff $\mathtt{p}(\mathrm{a})$ does not satisfy explicit integrity modulo endorsement, provided that $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ causes a security failure (contrapositive form).

If $\mathcal{R}(\mathtt{p}(\mathrm{a}))$ causes a security failure then $\mathtt{p}(\mathrm{a})$ is unsafe according to the faithfulness of $\mathcal{R}$ (Theorem 5.2.1). This implies that $\mathtt{p}(\mathrm{a}) \Downarrow \sigma\kappa$ where $\kappa = \mathtt{E}[\mathtt{u.sso}(\mathtt{v})]$, $\mathtt{v} = \mathtt{new\ C}(\bar{\mathtt{v}})$, $\mathtt{u} = \mathtt{new\ D}(\bar{\mathtt{u}})$ and shadow of v is $sv = \mathtt{shadow\ C}(\bullet, \overline{sv})$. Without loss of generality, let $\kappa$ be the first configuration in the execution trace with such structure and assume that $\mathtt{p}(\mathrm{a})$ satisfies explicit integrity modulo endorsement up to $\kappa$. Moreover, let $tail(\sigma) = \mathtt{e}$. Then we have

$$\mathtt{p}(\mathrm{a}) \rightarrow_f^* \mathtt{e} \overset{\overline{\mathrm{e}}}{\rightarrow}_{g_1}^* \kappa \xrightarrow{ie(\mathtt{v})}_{g_2} \mathtt{e}[\mathtt{v}/\mathtt{x}][\mathtt{u}/\mathtt{this}],$$

where $\overline{\mathrm{e}} \notin \mathbf{E_{en}}^*$ and $p_e(\mathrm{a}, \mathtt{e}_0, f) = p_e(\mathrm{a}, \mathtt{e}_0, g_1 * f)$ for initial expression $\mathtt{e}_0$ defined as $\mathtt{e}_0 =$ $\mathtt{new\ TopLevel().main(u')}$. That is, for any $\mathrm{a}'$, if $\pi_2(f(\mathrm{a}, \mathtt{e}_0)) = \pi_2(f(\mathrm{a}', \mathtt{e}_0))$ then $\pi_2(g_1 * f(\mathrm{a}, \mathtt{e}_0)) = \pi_2(g_1 * f(\mathrm{a}', \mathtt{e}_0))$.

So then in order to show that $\mathtt{p}(\mathrm{a})$ does not satisfy explicit integrity modulo endorsement, it should be studied whether $p_e(\mathrm{a}, \mathtt{e}_0, f) \neq p_e(\mathrm{a}, \mathtt{e}_0, g_2 * g_1 * f)$, i.e., whether there exists some attack $\mathrm{a}''$ such that $\pi_2(f(\mathrm{a}, \mathtt{e}_0)) = \pi_2(f(\mathrm{a}'', \mathtt{e}_0))$ and $\pi_2(g_2 * g_1 * f(\mathrm{a}, \mathtt{e}_0)) \neq \pi_2(g_2 * g_1 * f(\mathrm{a}'', \mathtt{e}_0))$. Since $\mathtt{v}$ is tainted, it is dependent on the user input $\mathrm{a}$ without being affected by any constant library method (otherwise, $\mathtt{v}$ was not tainted due to not being overtainted by those library methods). If there does not exist a different attack $\mathrm{a}''$ from $\mathrm{a}$ causing some different value $\mathtt{v}''$ from $\mathtt{v}$ to be passed to sso $\mathtt{D.m}$, then all attacks end in the same tainted value $\mathtt{v}$ at the sso $\mathtt{D.m}$. This is in contrast with constant library methods to propagate taint properly. Thus, there exist some attack $\mathrm{a}''$ from $\mathrm{a}$ causing some different value $\mathtt{v}''$ from $\mathtt{v}$ to be passed to sso $\mathtt{D.m}$, which completes the proof. □

## 5.4 Discussion

In this section, we exemplify how taint propagation instrumentation of library methods (in light of Definition 3.7.2 and Definition 3.7.2) affects soundness and completeness of integrity taint analysis.

To study the role of undertainting library methods assume we have defined taint propagation

$$\texttt{TopLevel.main(new String(}''\texttt{Hello}''\texttt{).sso(new String(}''\texttt{Hello}''\texttt{).concat(new String(}''\texttt{world}''\texttt{))))} =_\circ$$
$$\texttt{TopLevel.main(new String(}''\texttt{Attack}''\texttt{).sso(new String(}''\texttt{Attack}''\texttt{).concat(new String(}''\texttt{world}''\texttt{))))}$$

$$\texttt{TopLevel.main(new String(}''\texttt{Hello}''\texttt{).sso(new String(}''\texttt{Helloworld}''\texttt{)))} \neq_\circ$$
$$\texttt{TopLevel.main(new String(}''\texttt{Attack}''\texttt{).sso(new String(}''\texttt{Attackworld}''\texttt{)))}$$

$$\texttt{TopLevel.main(new String(}''\texttt{Hello}''\texttt{).sso(new String(}''\texttt{Helloworld}''\texttt{)))} =_\circ$$
$$\texttt{TopLevel.main(new String(}''\texttt{Attack}''\texttt{).sso(new String(}''\texttt{Attackworld}''\texttt{)))}$$

Figure 5.3: Undertainting Example.

policy $\mathrm{Prop}(\circ, \texttt{String.concat}(t_1, t_2))$ for library method `String.concat`. Let

$$mbody_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.sso}(\texttt{x.concat}(\texttt{new String}(''\texttt{world}''))).$$

Consider two attacks $\texttt{new String}(''\texttt{Hello}'')$ and $\texttt{new String}(''\texttt{Attack}'')$. Note that when `sso` is invoked for these two attacks states are trust equivalent, given as the first trust equivalence formula in Figure 5.3. But when library method returns value the states are not trust equivalent (second formula in Figure 5.3), since the primitive values are syntactically different but untainted in those states. Then neither of these attacks causes security failure, but explicit secrecy modulo endorsement is not satisfied, since attacks generate different events that refine attacker power: $ie(\texttt{new String}(''\texttt{Helloworld}''))$ and $ie(\texttt{new String}(''\texttt{Attackworld}''))$. Thus, soundness is not satisfied.

However, if the propagation policy for `String.concat` is defined as (3.1) in Section 3.6.1, then `String.concat` is not undertainting, since the states are trust equivalent after return. This is due to the fact that syntactically different primitive values are marked as tainted in those states (third formula in Figure 5.3). In this case, both attacks cause security failure and so soundness is satisfied.

To study effects of overtainting, specifically incompleteness, consider primitive operation $prim(str)$ and its OO wrapper `String.primwrapper` with the taint propagation policy given in Section 3.7. Assuming

$$mbody_{CT}(\mathtt{main}, \mathtt{TopLevel}) = \mathtt{x}, \mathtt{x.sso}(\mathtt{x.primwrapper}()),$$

any two attacks $\mathtt{new\ String}(str)$ and $\mathtt{new\ String}(str')$ cause security failure but the same integrity event $ie(\mathtt{new\ Int}(0))$ is generated since both execution traces end up invoking `sso` on $\mathtt{new\ Int}(0)$. This implies that the taint analysis for this program is not complete.

Since each primitive operation *Op* could be defined arbitrarily and the taint propagation policy predicated on $\mathrm{Prop}$ for the library method that wraps *Op* is user-defined, soundness and completeness of taint analysis depends on the accuracy of the user-defined taint propagation policy for each given primitive operation.

# Chapter 6

# Related Work

Secure information flow [25] and its interpretation as the well-known hyperproperty [26] of noninterference [5] is challenging to implement in practical settings [4] due to implicit flows. Taint analysis is thus an established solution to enforce confidentiality and integrity policies since it tracks only direct data flow control. Various systems have been proposed for both low and high level level languages. The majority of previous work, however, has been focused on taint analysis policy specification and enforcement (e.g. [6, 7, 27, 28]), rather than capturing the essence of direct information flow which could provide an underlying framework to study numerous taint analysis tools.

Knowledge-based semantics has been introduced by Askarov et al. [19] as a general model for information flow of confidential data, concentrated on cryptographic computations and key release (declassification [29]) and later employed in other data secrecy analyses [30, 31, 16]. Recently, Schoepe et al. [8] have proposed the semantic notion of correctness for taint tracking that enforces confidentiality policies of direct information flow, called explicit secrecy. To this end, they proposes a knowledge-based semantics, influenced by Volpano's weak secrecy [32] and gradual release [19]. Explicit secrecy is defined as a property of a program, where the program execution does not change the explicit knowledge of public user. The authors show that noninterference is not comparable to explicit secrecy. However, since attacker knowledge does not capture the attacker model and nature of problem in integrity flows, explicit secrecy cannot be applied for direct flow of data integrity. For this purpose, we propose explicit integrity. Moreover, rather than restricting the discussion to direct information flow in a low level language, we also model a high level OO language with a functional flavor to represent generality of our framework.

A counterpart for attacker knowledge in the realm of general flow of information integrity, called attacker power [16], is introduced as the set of low integrity inputs that generate the same observables. In this regard, Askarov et al. [16] use holes in the syntax of program code for injection points, influenced by [33]. However, their attack model is different as the low integrity and low confidentiality user is able to inject program code in the main program, by which she could gain more knowledge. We have tailored attacker power for explicit flows using state transformers, in order to interpret integrity taint analysis.

Birgisson et al. [14] give a unified framework to capture different flavors of integrity, in particular integrity via information flow and via different types of invariance. Similar to other works in this line, they give a simple imperative language with labeled operational semantics in order to enforce integrity policies through communication with a monitor. In contrast, we use program rewriting techniques to enforce policies regarding flow of data integrity, which are applicable to legacy systems.

The work presented here partially relies on our previous work [10], in which we propose a policy language and enforcement technique for in-depth dynamic integrity taint analysis based on a well-developed formal foundation explored in [23]. Enforcement of taint policies formalizes Phosphor [11, 12], which is an attempt to apply taint tracking more generally in Java to any primitive type and object class. To the best of our knowledge, other Java-based techniques do not enjoy the generality that Phosphor provides for all Java data types and classes. For instance, Chin et al. [17] and Haldar et al. [18] only focus on specific classes as user inputs, and in contrast to our work, *only* strings are endowed with integrity information, whereas all values are assigned integrity labels in our approach.

# Chapter 7

# Conclusion

This work proposes a semantic framework to model direct flow of data integrity enforced by integrity taint analysis techniques. In contrast to previous work, this framework is general enough to model direct flows in both imperative and functional settings and could be used to study the prerequisites for correct taint propagation. We have used this framework to extract such requirements for a general purpose taint tracking tool in Java and prove its soundness and completeness. This exemplifies the use of the proposed semantics to establish provable correctness conditions for a rewriting algorithm that instruments integrity taint analysis in the presence of input sanitization. A rewriting approach supports development of tools that can be applied to legacy code without modifying language implementations.

This work provides an underlying semantic framework to study numerous other integrity taint analyzers in the future.

**Acknowledgement** This work is supported in part by the National Science Foundation under Grant No. 1408801.

# Bibliography

[1] Benjamin Livshits, Michael Martin, and Monica S Lam. Securifly: Runtime protection and recovery from web application vulnerabilities. Technical report, Technical report, Stanford University, 2006.

[2] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.

[3] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *Operating Systems Review*, 45(1):142–154, 2011.

[4] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[5] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE S&P*, pages 11–20, 1982.

[6] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, pages 317–331, 2010.

[7] Benjamin Livshits. Dynamic taint tracking in managed runtimes. Technical report, Technical Report MSR-TR-2012-114, Microsoft Research, 2012.

[8] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE EuroS&P*, pages 15–30, 2016.

[9] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[10] Sepehr Amir-Mohammadian and Christian Skalka. In-depth enforcement of dynamic integrity taint analysis. In *PLAS*, 2016.

[11] Jonathan Bell and Gail E. Kaiser. Phosphor: illuminating dynamic data flow in commodity jvms. In *OOPSLA*, pages 83–101, 2014.

[12] Jonathan Bell and Gail E. Kaiser. Dynamic taint tracking for java with phosphor (demo). In *ISSTA*, pages 409–413, 2015.

[13] Juan José Conti and Alejandro Russo. A taint mode for Python via a library. In *NordSec*, pages 210–222, 2010.

[14] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *ICISS*, pages 48–65, 2010.

[15] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.

[16] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *ESOP*, pages 64–84, 2010.

[17] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *ACM SWS*, pages 3–12, 2009.

[18] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC*, pages 303–311, 2005.

[19] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE S&P*, pages 207–221, 2007.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[21] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (And never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[22] Ulf Nilsson and Jan Maluszyynski. Definite logic programs. In *Logic, Programming and Prolog*, chapter 2. 2000.

[23] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Correct audit logging: Theory and practice. In *POST*, pages 139–162, 2016.

[24] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

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

[26] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[27] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, pages 385–398, 2013.

[28] Zheng Wei and David Lie. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *SPSM Workshop at CCS*, pages 27–38, 2014.

[29] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[30] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, 2008.

[31] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.

[32] Dennis M. Volpano. Safety versus secrecy. In *SAS*, pages 303–311, 1999.

[33] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.