Maybe Tainted Data: Theory and a Case Study

Christian Skalka [a,*], Sepehr Amir-Mohammadian [b], and Samuel Clark [a]
[a] *University of Vermont, Department of Computer Science*
E-mails: ceskalka@uvm.edu, samuel.clark@uvm.edu
[b] *University of the Pacific, Department of Computer Science*
E-mail: samirmohammadian@pacific.edu

Abstract. Dynamic taint analysis is often used as a defense against low-integrity data in applications with untrusted user interfaces. An important example is defense against XSS and injection attacks in programs with web interfaces. Data sanitization is commonly used in this context, and can be treated as a precondition for endorsement in a dynamic integrity taint analysis. However, sanitization is often incomplete in practice. We develop a model of dynamic integrity taint analysis for Java that addresses imperfect sanitization with an in-depth approach. To avoid false positives, results of sanitization are endorsed for access control (aka prospective security), but are tracked and logged for auditing and accountability (aka retrospective security).

We show how this heterogeneous prospective/retrospective mechanism can be specified as a uniform policy, separate from code. We then use this policy to establish correctness conditions for a program rewriting algorithm that instruments code for the analysis. These conditions synergize our previous work on the semantics of audit logging with *explicit integrity* which is an analogue of noninterference for taint analysis. A technical contribution of our work is the extension of explicit integrity to a high-level functional language setting with structured data, vs. previous systems that only address low level languages with unstructured data. Our approach considers endorsement which is crucial to address sanitization. An implementation of our rewriting algorithm is presented that hardens the OpenMRS medical records software system with in-depth taint analysis, along with an empirical evaluation of the overhead imposed by instrumentation. Our results show that this instrumentation is practical.

Keywords: Auditing, Dynamic taint analysis, Program rewriting

1. Introduction

Dynamic taint analysis implements a “direct” or “explicit” information flow analysis to support a variety of security mechanisms [1]. Similar to information flow, taint analysis can be used to support either confidentiality or integrity properties. An important application of integrity taint analysis is to prevent the execution of security sensitive operations on untrusted data, in particular to combat cross-site scripting (XSS) and SQL injection attacks in web applications [2]. Any untrusted user input is marked as tainted, and then taint is tracked and propagated through data flow to ensure that tainted data is not used by security sensitive operations.

Of course, since web applications aim to be interactive, user input is needed for certain security sensitive operations such as database calls. To combat this, *sanitization* is commonly applied in practice to analyze and possibly modify data. From a taint analysis perspective, sanitization is a precondition

[*]Corresponding author. E-mail: ceskalka@uvm.edu.

for integrity endorsement, i.e. subsequently viewing sanitization results as high integrity data. However, while sanitization is usually endorsed as "perfect" by taint analysis, in fact it is not. Indeed, previous work has identified a number of flaws in existing sanitizers in a variety of applications [2, 3], including the exploits to inject commands in web application, and the work here is in fact inspired by discovery of an XSS attack vector in the OpenMRS medical records software systems due to incomplete sanitization, discussed below in Section 1.1. We have demonstrated the practicality of exploiting the discovered XSS vulnerability in OpenMRS. We call such incomplete sanitizers *partially trusted* or *imperfect* throughout the paper.

Thus, a main challenge we address is how to mitigate imperfect sanitization in taint analysis. An important feature of our approach is an *in-depth* [4] security policy, that combines the typical blocking (prospective) behavior of taint-based access control with audit logging (retrospective) features. In the presence of imperfect sanitization, this allows false positives to be avoided, while still providing retrospective security measures via audit logs in case of attacks that leverage this imperfection. We are concerned with both efficiency and correctness– we develop a language model intended to capture the essence of Phosphor [5, 6], an existing Java taint analysis system with empirically demonstrated efficiency. To be applicable to legacy systems such as OpenMRS, we propose a program rewriting approach, Our program rewriting algorithm takes as input a heterogeneous (prospective and retrospective) taint analysis policy specification and input code, and instruments the code to support the policy. The policy allows user specification of taint sources, secure sinks, and sanitizers. A distinct feature of our system is that results of sanitization are considered "maybe tainted" data, which are allowed to flow into security sensitive operations but in such cases are entered in a log to support auditing and accountability.

A contribution of our approach is a uniform expression of an in-depth security policy that combines prospective (taint analysis) and retrospective (audit logging) policy features, and proof that our rewriting algorithm enforces this policy. To characterize retrospective correctness we leverage our previous work on the the semantics of retrospective security [7]. To characterize prospective correctness, we aim to go deeper than operational definitions [1, 8] and characterize correctness as a higher level semantic property, in particular as a hyprerproperty [9]. To this end we propose a semantic framework called *explicit integrity*, that is an extension of explicit secrecy [10] to a high-level (Java) language model with structured data. Explicit integrity integrity is analagous to (integrity) noninterference, but applies to taint analysis that is concerned only with direct aka explicit information flows. Intuitively, in programs that enjoy explicit integrity, low-integrity (tainted) data does not flow directly into high-integrity sinks during program execution. Both explicit secrecy and explicit integrity are defined independently of language-level instrumentation, and (like noninterference) are hyperproperties as formulated in [10] and in this work. Furthermore, we consider the variant explicit integrity *modulo endorsement*, since endorsement is necessary in the taint analysis to accurately reflect the results of sanitization.

1.1. Practical Motivations

While our work is based on formal foundations it is inspired by practical concerns, in particular a security flaw we discovered in a previous version (2.4) of OpenMRS originally reported in [11][1]. This flaw allows an attacker to launch persistent XSS attacks. When a web-based software receives and stores user input without proper sanitization, and later retrieves this information for (other) users, persistent XSS attacks could take place.

[1]We responsibly disclosed the vulnerabilities we found to the OpenMRS development community, and they have been corrected in current versions.

OpenMRS uses a set of validators to enforce expected data formats by implementation of the `Validator` interface (e.g., `PersonNameValidator`, `VisitTypeValidator`, etc.). For some of these classes the implementation is strict enough to reject script tags by enforcing data to match a particular regular expression, e.g., `PersonNameValidator`. However, `VisitTypeValidator` lacks such restriction and only checks for object fields to avoid being null, empty or whitespace, and their lengths to be correct. Thus the corresponding webpage that receives user inputs to construct `VisitType` objects (named `VisitTypeForm.jsp`) is generally not able to perform proper sanitization through the invocation of the validator implemented by `VisitTypeValidator`. A `VisitType` object is then stored in the MySQL database, and could be retrieved later based on user request. For instance, `VisitTypeList.jsp` queries the database for all defined `VisitType` objects, and sends `VisitType` names and descriptions to the client side. Therefore, the attacker can easily inject scripts as part of `VisitType` name and/or description, and the constructed object would be stored in the database and possibly in a later stage retrieved and executed in the victim's client environment.

Integrity taint tracking is a well-recognized solution against these sorts of attacks that deals direct information flow analysis (and hence explicit integrity). In our example, using taint analysis the tainted `VisitType` object would be prevented from retrieval and execution. The addition of sanitization methods would also be an obvious step, and commensurate with an integrity taint analysis approach– sanitized objects would be endorsed for the purposes of prospective security. However, many attack scenarios demonstrate degradation of taint tracking effectiveness due to unsound or incomplete input sanitization [2, 3]. Hence our introduction of "maybe tainted" data, which is allowed to flow into security sensitive operations but in such cases are entered in a log to support auditing and accountability.

1.2. The Security and Threat Model

The security problem we consider is about the integrity of data being passed to security sensitive operations (SSOs) in a direct manner. An important example is a string entered by an untrusted user 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.

In contrast to standard information flow which is concerned with both direct (aka explicit) and indirect (aka implicit) flows, taint analysis is only concerned with direct flow. Direct flows transfer data directly between variables, e.g., n_1 and n_2 directly affect the result of $n_1 + n_2$. Indirect flows are realized when data can affect the result of code dispatch– the standard example is a conditional expression **if** v **then** e_1 **else** e_2 where the data v indirectly affects the valuation of the expression by guarding dispatch.

More precisely, we posit that top-level programs $\mathtt{p}$ in this security setting are parameterized by a low integrity data source **a**, and an arbitrary number of secure sinks, aka security sensitive oprations (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 *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 *SSOs* of the same form. 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. Explicit integrity, as a high-level property, is instantiated for this model.

We assume that our program rewriting algorithm is trusted. Input code is trusted to be not malicious, though it may contain errors. We note that this assumption is important for application of taint analysis that disregards indirect flows, since there is confidence that the latter will not be actively exploited by non-malicious code. We assume that untrusted data sources provide low integrity data, though in this work we only consider tainted "static" values, e.g., strings, not tainted code that may be run as part of the main program execution. However, the latter does not preclude hardening against XSS or injection attacks in practice, if we consider an evaluation method to be an SSO.

1.3. Overview by Example

As a running example for the paper, consider the following code, where we imagine a tainted input string `"hello"` is concatenated with an untainted string `"world"`, and then sanitized by a method `sanitize` of a `Sec` object that implements security functionality. The sanitized result is then passed to an *sso* called `secureMeth`, also in the `Sec` class.

```
new Sec().secureMeth(new Sec().sanitize("hello".concat(new String("world"))))
```

In order to define the logging policy, we will define an operational (trace) semantics of programs where direct information flow is defined as a property of traces. This property correlates taint labels with values in traces– in the above expression the tainted label $\bullet$ is correlated with `"hello"`, the untainted label $\circ$ is correlated with `"world"`. And in the trace of this programs execution, the tainted label $\bullet$ is correlated with the concatenated string `"hello world"`, due to the propagation of taint. Sanitization is typically associated with endorsement in taint tracking systems [10]. However, since sanitization can often only be partially trusted, we propose to consider such santiziation results to be "maybe tainted", which is indicated by correlation with a maybe tainted label $\odot$. The logging policy should then specify that maybe tainted data entering any *sso* should be allowed, but logged.

In all cases, correlation of labels with values in structured data is accomplished via "shadows" of expressions, which are shape-conformant with source language expressions and carry taint lables. For example, in the trace of the above expression, the result of sanitization is the following:

$$\texttt{TopLevel.main}\big(\texttt{new Sec().secureMeth(new String("hello world"))}\big)$$

This expression has the following shadow:

$$\texttt{TopLevel.main}\big(\texttt{shadow Sec}(\circ)\texttt{.secureMeth(shadow String}(\odot,\delta))\big)$$

Note that the shadow replaces distinct lexical values with a dummy value δ, but can be "overlaid" on the expression to obtain the tain correlation. Thus, our logging policy would specify that the maybe tained string `"hello world"` would be allowed to enter the *sso* `secureMeth`, but logged. This example is revisited later in Example 3.2, where we go into detail about its evaluation, and its shadow expressions that capture the operational semantics of taint analysis.

Subsequently, we develop a rewriting algorithm called *Phos* that instruments programs to implement taint analysis, as well as to implement logging of maybe tainted data, that is correct with respect to the shadow specification. We revisit this example again in 4.1.4 to show how taint labels, taint propagation, and logging of maybe tainted data is made explicit by *Phos*.

1.4. Technical Overview

The technical development of the paper proceeds as follows. In Section 2 we describe a formal semantics of auditing, and the conditions for correctness of audit rewriting algorithms. That is, we define what it means for a program instrumentation to correctly log information. In Section 2.1, we introduce information algebra [12] as the basis of our model for correct audit log generation. We characterize logging specifications and correctness conditions for audit logs in a high-level manner using information algebra, and show how information elements and operations can be instantiated using first order logic.

In Section 3 we develop a source language model based on featherweight Java (Section 3.1), called FJ. We show how to logically specify an in-depth taint analysis policy separately from code in Section 4 via safety property and logging specifications. In Section 3.2 we develop a target language model $\mathrm{FJ}_{\mathrm{taint}}$ with instrumentation for operationally enforcing an in-depth taint analysis, which we show to be correct according to our formal condition in Section 4.2, with our main result being Theorem 4.1.

While Theorem 4.1 establishes correctness conditions for information in audit logs in an operational sense, Section 5 focuses on the high level security property of dynamic integrity taint analysis that is tailored for direct information flow. In Section 5.4, we show that our enforcement mechanism satisfies the hyperproperty of explicit integrity modulo endorsement (Theorem 5.1). In Section 6 we discuss our implementation of the in-depth taint analysis specification presented in Section 4 for the OpenMRS medical records system. We also describe experiments for empirical evaluation of this implementation and discuss results. In Section 7 we discuss related work and conclude the paper.

For the sake of brevity, we have omitted the proofs of all Lemmas and Theorems. Readers are referred to our accompanying Technical Report [13] for these details in full.

2. Foundations for In-Depth Policy Specification

In this section we establish formal foundations for a semantics of prospective and retrospective policy features. and the correctness of audit instrumentation. An appeal of our approach is that both safety properties [14] and logging correctness can be formulated, so we are able to uniformly characterize operational correctness conditions for in-depth integrity taint analysis in our framework. This framework was initially developed in previous work [7] where we studied so-called “break-the-glass” policies for medical records software. In that work we justified the generality of our framework and discuss its details at length. Here we reiterate the main technical points of the framework to allow a standalone formal presentation.

We leverage ideas from the theory of *information algebra* [12, 15], which is an abstract mathematical framework for information systems. In short, we interpret program traces as information, and logging specifications as functions from traces to information. This separates logging specifications from their implementation in code, and defines exactly the information that should be in an audit log. This in turn establishes correctness conditions for audit logging implementations.

2.1. Introduction to Information Algebra

Information algebra is an algebraic theory of information where information is seen as a collection of *information elements* with fundamental aggregation and refinement operations. The algebra consists of two domains, an information domain and a query domain. The information domain Φ is the set of information elements that can be aggregated in order to build more inclusive information elements. The query domain E is a lattice of querying sublanguages in which the partial order relation among these sublanguages represents the granularity of the queries. The information and query domains are left abstract in the general theory– instantiation examples include relational algebra and first order logic as we discuss below. By definition any instantiation must include basic operations for combining information and for focusing on components of information.

Definition 2.1. *Any information algebra (Φ, E) includes two basic operators:*

- *Combination $\otimes : \Phi \times \Phi \to \Phi$: The operation $X \otimes Y$* combines (or, aggregates) *the information in elements $X, Y \in \Phi$.*
- *Focusing $\Rightarrow$: $\Phi \times E \to \Phi$: The operation $X^{\Rightarrow S}$ isolates the elements of $X \in \Phi$ that are relevant to a* sublanguage *$S \in E$, i.e. the subpart of X specified by S.*

Using the combination operator we can define a partial order relation on Φ to compare the information contained in the elements of Φ. A partial ordering is induced on Φ by the so-called *information ordering* relation $\leqslant$, where intuitively for $X, Y \in \Phi$ we have $X \leqslant Y$ iff Y contains at least as much information as X, though its precise meaning depends on the particular algebra.

Definition 2.2. *X is contained in Y, denoted as $X \leqslant Y$, for all $X, Y \in \Phi$ iff $X \otimes Y = Y$.*

Definition 2.3. *We say that X and Y are* information equivalent*, and write $X = Y$, iff $X \leqslant Y$ and $Y \leqslant X$.*

For a more detailed account of information algebra, the reader is referred to a definitive survey paper [15].

2.1.1. Illustrative Example: Relational Algebras

Relational algebra is a well-recognized instance of information algebra. The formulation of relation algebra as an information algebra by Kohlas [28] is an illustrative example of this information theoretic framework, given in the following.

Let $\mathcal{A}$ denote the set of *attributes*, $\mathcal{A}_i \subseteq \mathcal{A}$ for $i \in \{1,2,3\}$, $\mathcal{A}_2 \subseteq \mathcal{A}_1$, and assume that $\mathcal{A}_1 = \{a_1, ..., a_n\}$. Each tuple $((a_1 : x_1), \cdots, (a_n : x_n))$ can be formulated as a function $f : \mathcal{A}_1 \to \{x_1, ..., x_n\}$, where $f(a_i) = x_i$. x_is are values from potentially different domains.

Function $f[\mathcal{A}_2] : \mathcal{A}_2 \to \{x_1, ..., x_n\}$ is the *restriction* of f to $\mathcal{A}_2$, defined as $f[\mathcal{A}_2](a) = f(a)$, for all $a \in \mathcal{A}_2$. A *relation* R over $\mathcal{A}_1$ is a set of functions f defined on a specific set of attributes $\mathcal{A}_1$. Then, the *projection* of R on $\mathcal{A}_2$ is defined as $\pi_{\mathcal{A}_2}(R) = \{f[\mathcal{A}_2] \mid f \in R\}$. The *natural join* of relation R over $\mathcal{A}_1$ and R' over $\mathcal{A}_3$ is defined as $R \bowtie R' = \{f \mid dom(f) = \mathcal{A}_1 \cup \mathcal{A}_3, f[\mathcal{A}_1] \in R, f[\mathcal{A}_3] \in R'\}$.

Instantiation. Let $\mathbb{R}$ be the universe of all relations R. Then, $(\mathbb{R}, \mathcal{P}(\mathcal{A}))$ is an information algebra with following definitions for combination and focusing:

$$R \otimes R' \triangleq R \bowtie R' \qquad\qquad R^{\Rightarrow \mathcal{A}_1} \triangleq \pi_{\mathcal{A}_1}(R)$$

Note that in this formulation, the restriction operator is defined partially on the set of attributes [28].

According to Definition 2.2, for all relations R and R', $R \leqslant R'$ iff $R \bowtie R' = R'$. For example, $\pi_{\mathcal{A}_1}(R) \leqslant R$. Moreover, the set of querying sublanguages $\mathcal{P}(\mathcal{A})$ is a lattice induced by subset containment relation $\subseteq$.

2.2. A General Model for Logging Specifications

Following [14], an *execution trace* $\tau = \kappa_0\kappa_1\kappa_2\ldots$ is a possibly infinite sequence of configurations κ that describe the state of an executing program. We deliberately leave configurations abstract, but examples abound and we explore a specific instantiation for FJ-based calculus in Section 3. Note that an execution trace τ may represent the partial execution of a program, i.e. the trace τ may be extended with additional configurations as the program continues execution. We use metavariables τ and σ to range over traces, and use $\varnothing$ to denote an empty trace.

We assume a given function $\lfloor\cdot\rfloor$ that is an injective mapping from traces to Φ. This mapping *interprets a given trace as information*, where the injective requirement ensures that information is not lost in the interpretation. For example, if σ is a proper prefix of τ and thus contains strictly less information, then formally $\lfloor\sigma\rfloor \leqslant \lfloor\tau\rfloor$. We intentionally leave both Φ and $\lfloor\cdot\rfloor$ underspecified for generality, though application of our formalism to a particular logging implementation requires instantiation of them.

We let LS range over *logging specifications*, which are functions from traces to Φ. As for Φ and $\lfloor\cdot\rfloor$, we intentionally leave the language of specifications abstract, but consider a particular instantiation in Section 2.6. Intuitively, $LS(\tau)$ denotes the information that should be recorded in an audit log during the execution of τ given specification LS, regardless of whether τ actually records any log information, correctly or incorrectly. We call this the semantics of the logging specification LS.

We assume that auditing is implementable, requiring at least that all conditions for logging any piece of information must be met in a finite amount of time. As we will show, this restriction implies that correct logging instrumentation is a safety property [14].

Definition 2.4. *We require of any logging specification LS that for all traces τ and information $X \leqslant LS(\tau)$, there exists a finite prefix σ of τ such that $X \leqslant LS(\sigma)$.*

It is crucial to observe that some logging specifications may *add* information not contained in traces to the auditing process. Security information not relevant to program execution (such as ACLs), interpretation of event data (statistical or otherwise), etc., may be added by the logging specification. For example, in the OpenMRS system [16], logging of sensitive operations includes a human-understandable "type" designation which is not used by any other code. Thus, given a trace τ and logging specification LS, it is *not* necessarily the case that $LS(\tau) \leqslant \lfloor\tau\rfloor$. Audit logging is not just a filtering of program events.

2.3. Correctness Conditions for Audit Logs

A logging specification defines what information should be contained in an audit log. In this section we develop formal notions of *soundness* and *completeness* as audit log correctness conditions. We use metavariable $\mathbb{L}$ to range over audit logs. Again, we intentionally leave the language of audit logs unspecified, but assume that the function $\lfloor\cdot\rfloor$ is extended to audit logs, i.e. $\lfloor\cdot\rfloor$ is an injective mapping from audit logs to Φ. Intuitively, $\lfloor\mathbb{L}\rfloor$ denotes the information in $\mathbb{L}$, interpreted as an element of Φ.

An audit log $\mathbb{L}$ is sound with respect to a logging specification LS and trace τ if the log information is contained in $LS(\tau)$. Similarly, an audit log is complete with respect to a logging specification if it contains all of the information in the logging specification's semantics. Crucially, both definitions are independent of the implementation details that generate $\mathbb{L}$.

Definition 2.5. *Audit log $\mathbb{L}$ is* sound with respect to logging specification LS and execution trace τ *iff* $\lfloor\mathbb{L}\rfloor \leqslant LS(\tau)$.

Definition 2.6. *Audit log* $\mathbb{L}$ *is* complete with respect to logging specification *LS* and execution trace τ *iff* $LS(\tau) \leqslant \lfloor \mathbb{L} \rfloor$.

2.4. Correct Logging Instrumentation is a Safety Property

In case program executions generate audit logs, we write $\tau \rightsquigarrow \mathbb{L}$ to mean that trace τ generates $\mathbb{L}$, i.e. $\tau = \kappa_0 \dots \kappa_n$ and $logof(\kappa_n) = \mathbb{L}$ where $logof(\kappa)$ denotes the audit log in configuration κ, i.e. the residual log after execution of the full trace. Ideally, information that *should* be added to an audit log, *is* added to an audit log, immediately as it becomes available. Using the term "instrumentation" to refer to program elements for audit log generation, this idea is formalized as follows.

Definition 2.7. *For all logging specifications LS, the trace* τ *is* ideally instrumented for *LS iff for all finite prefixes* σ *of* τ *we have* $\sigma \rightsquigarrow \mathbb{L}$ *where* $\mathbb{L}$ *is sound and complete with respect to LS and* σ.

We observe that the restriction imposed on logging specifications by Definition 2.4, implies that ideal instrumentation of any logging specification is a safety property in the sense defined by Schneider [14].

Theorem 2.1. *For all logging specifications LS, the set of ideally instrumented traces is a safety property.*

This result implies that e.g. edit automata can be used to enforce instrumentation of logging specifications [17]. However, theory related to safety properties and their enforcement by execution monitors [14, 18] does not provide an adequate semantic foundation for audit log generation, nor an account of soundness and completeness of audit logs.

2.5. Implementing Logging Specifications with Program Rewriting

The above-defined correctness conditions for audit logs provide a foundation on which to establish correctness of logging implementations. Here we consider program rewriting approaches. Since rewriting concerns specific languages, we introduce an abstract notion of programs $\mathbf{p}$ with an operational semantics that can produce a trace. We write $\mathbf{p} \Downarrow \sigma$ iff program $\mathbf{p}$ can produce execution trace τ, either deterministically or non-deterministically, and σ is a *finite* prefix of τ.

A rewriting algorithm $\mathcal{R}$ is a (partial) function that takes a program $\mathbf{p}$ in a source language and a logging specification *LS* and produces a new program, $\mathcal{R}(\mathbf{p}, LS)$, in a target language.[2] The intent is that the target program is the result of instrumenting $\mathbf{p}$ to produce an audit log appropriate for the logging specification *LS*. A rewriting algorithm may be partial, in particular because it may only be intended to work for a specific set of logging specifications.

Ideally, a rewriting algorithm should preserve the semantics of the program it instruments. That is, $\mathcal{R}$ is semantics-preserving if the rewritten program simulates the semantics of the source code, modulo logging steps. We assume given a correspondence relation $\cong$ on execution traces. A coherent definition of correspondence should be similar to a bisimulation, but is not necessarily a bisimulation, since the instrumented target program may be in a different language than the source program. We deliberately leave the correspondence relation underspecified, as its definition will depend on the instantiation of the

[2]We use metavariable $\mathbf{p}$ to range over programs in either the source or target language; it will be clear from context which language is used.

model. Possible definitions are that traces produce the same final value, or that traces when restricted to a set of memory locations are equivalent up to stuttering (i.e., different numbers of "internal" execution steps that do not affect memory). Furthermore, because rewriting will often add blocking checks for unsafe behaviors (as in the case we will study), semantics preservation is defined up to simulation of sets of program traces that will typically be defined as a safety property. We provide a definition of correspondence for FJ-calculus source and target languages in Section 4.2, that illustrates these concepts.

Definition 2.8. *Let T be a set of program traces. Rewriting algorithm $\mathcal{R}$ is* semantics preserving up to *T iff for all programs $\mathbf{p}$ and logging specifications LS such that $\mathcal{R}(\mathbf{p}, LS)$ is defined, all of the following hold:*

(1) *For all traces $\tau \in T$ such that $\mathbf{p} \Downarrow \tau$ there exists τ' with $\tau \cong \tau'$ and $\mathcal{R}(\mathbf{p}, LS) \Downarrow \tau'$.*
(2) *For all traces τ such that $\mathcal{R}(\mathbf{p}, LS) \Downarrow \tau$ there exists a trace $\tau' \in T$ such that $\tau' \cong \tau$ and $\mathbf{p} \Downarrow \tau'$.*

In addition to preserving program semantics, a correctly rewritten program constructs a log in accordance with the given logging specification. More precisely, if LS is a given logging specification and a trace τ describes execution of a source program, rewriting should produce a program with a trace τ' that corresponds to τ (i.e., $\tau \cong \tau'$), where the log $\mathbb{L}$ generated by τ' ideally contains the same information as $LS(\tau)$. Some definitions of $\cong$ may allow several target-language traces to correspond to source-language traces. Hence we write $\mathit{simlogs}(\mathbf{p}, \tau)$ to denote a nonempty set of logs $\mathbb{L}$ such that, given source language trace τ and target program $\mathbf{p}$, there exists some trace τ' where $\mathbf{p} \Downarrow \tau'$ and $\tau \cong \tau'$ and $\tau' \rightsquigarrow \mathbb{L}$. The name *simlogs* evokes the relation to logs resulting from simulating executions in the target language.

The following definitions then establish correctness conditions for rewriting algorithms in conjunction with semantics preservation. Like semantics preservation, we define soundness and completeness with respect to a given set of traces.

Definition 2.9. *Let T be a set of traces. Rewriting algorithm $\mathcal{R}$ is* sound up to *T iff for all programs $\mathbf{p}$, logging specifications LS, and finite traces $\tau \in T$ where $\mathbf{p} \Downarrow \tau$, for all $\mathbb{L} \in \mathit{simlogs}(\mathcal{R}(\mathbf{p}, LS), \tau)$ it is the case that $\mathbb{L}$ is sound with respect to LS and τ.*

Definition 2.10. *Let T be a set of traces. Rewriting algorithm $\mathcal{R}$ is* complete up to *T iff for all programs $\mathbf{p}$, logging specifications LS, and finite traces $\tau \in T$ where $\mathbf{p} \Downarrow \tau$, for all $\mathbb{L} \in \mathit{simlogs}(\mathcal{R}(\mathbf{p}, LS), \tau)$ it is the case that $\mathbb{L}$ is complete with respect to LS and τ.*

Note also that without semantics preservation, soundness and completeness could be satisfied trivially in case $\mathit{simlogs}(\mathcal{R}(\mathbf{p}, LS), \tau)$ is empty.

2.6. A First Order Logic (FOL) Specification Language

Logics have been used in several well-developed auditing systems [19, 20], for the encoding of both audit logs and queries. FOL in particular is attractive due to readily available implementation support, e.g. Datalog and Prolog. We have shown in previous work that FOL is an information algebra, and useful for e.g. break the glass policy specification [7]. Here we summarize important definitions for the remainder of this paper.

Let Greek letters ϕ and ψ range over FOL formulas and let capital letters X, Y, Z range over sets of formulas. We posit a sound and complete proof theory supporting judgements of the form $X \vdash \phi$. In this text we assume without loss of generality a natural deduction proof theory.

Elements of our algebra are sets of formulas closed under logical entailment. Intuitively, given a set of formulas X, the closure of X is the set of formulas that are logically entailed by X, and thus represents all the information contained in X. In spirit, we follow the treatment of sentential logic as an information algebra explored in related foundational work [12], however our definition of closure is syntactic, not semantic.

Definition 2.11. *We define a closure operation C, and a set Φ_{FOL} of closed sets of formulas:*

$$C(X) = \{\phi \mid X \vdash \phi\} \qquad\qquad \Phi_{FOL} = \{X \mid C(X) = X\}$$

Note in particular that $C(\varnothing)$ is the set of logical tautologies.

Let *Preds* be the set of all predicate symbols, and let $S \subseteq \mathit{Preds}$ be a set of predicate symbols. We define *sublanguage* L_S to be the set of well-formed formulas over predicate symbols in S (including boolean atoms *true* and *false*, and closed under the usual first-order connectives and binders). We will use sublanguages to define refinement operations in our information algebra. Subset containment induces a lattice structure, denoted $\mathcal{S}$, on the set of all sublanguages, with $\mathcal{F} = L_{Preds}$ as the top element.

Now we can define the focusing and combination operators, which are the fundamental operators of an information algebra. Focusing isolates the component of a closed set of formulas that is in a given sublanguage. Combination closes the union of closed sets of formulas. Intuitively, the focus of a closed set of formulas X to sublanguage L is the refinement of the information in X to the formulas in L. The combination of closed sets of formulas X and Y combines the information of each set.

Definition 2.12. *Define:*

(1) *Focusing: $X^{\Rightarrow S} = C(X \cap L_S)$ where $X \in \Phi_{FOL}$, $S \subseteq \mathit{Preds}$*
(2) *Combination: $X \otimes Y = C(X \cup Y)$ where $X, Y \in \Phi_{FOL}$*

Properties of the algebra ensure that $\leqslant$ is a partial ordering by defining $X \leqslant Y$ iff $X \otimes Y = Y$, which in the case of our logical formulation means that for all $X, Y \in \Phi_{FOL}$ we have $X \leqslant Y$ iff $X \subseteq Y$, i.e. $\leqslant$ is subset inclusion over closed sets of formulas.

The following Theorem establishes that the construction is an information algebra– for a complete proof the reader is directed to [17].

Theorem 2.2. *Structure $(\Phi_{FOL}, \mathcal{S})$ with focus operation $X^{\Rightarrow S}$ and combination operation $X \otimes Y$ forms an information algebra.*

In addition, to interpret traces and logs as elements of this algebra, i.e. to define the function $\lfloor\cdot\rfloor$, we assume existence of a function $\mathit{toFOL}(\cdot)$ that injectively maps traces and logs to sets of FOL formulas, and then take $\lfloor\cdot\rfloor = C(\mathit{toFOL}(\cdot))$. To define the range of $\mathit{toFOL}(\cdot)$, that is, to specify how trace information will be represented in FOL, we assume the existence of *configuration description predicates P* which are each at least unary. Each configuration description predicate fully describes some element of a configuration κ, and the first argument is always a natural number n, indicating the time at which the configuration occurred. A set of configuration description predicates with the same timestamp describes a configuration, and traces are described by the union of sets describing each configuration in the trace. We will fully define $\mathit{toFOL}(\cdot)$ when we discuss particular source and target languages for program rewriting.

Formally, we define logging specifications in a logic programming style by using combination and focusing. Any logging specification is parameterized by a sublanguage S that identifies the predicate(s) to be resolved and Horn clauses X that define it/them, and can be defined via the functional *spec* from pairs (X, S) to specifications *LS*, where we use λ as a binder for function definitions in the usual manner:

Definition 2.13. *The function spec is given a pair* (X, S) *and returns a* FOL logging specification*, i.e. a function from traces to elements of* Φ_{FOL}*:*

$$spec(X, S) = \lambda\tau.(\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow S}.$$

We will formulate a particular example of a logging specification for maybe tainted data in Definition 3.3.

3. Direct Information Flow: Dynamic Integrity Taint Analysis

In this section we present a basic object-oriented calculus as the foundation of our language model. We also show how the in-depth integrity taint analysis model described in Section 1.2 can be specified as a logical property of program traces in this model, independent of program instrumentation. This allows us to define retrospective taint analysis as a logging specification in the style introduced in Section 2. Subsequently in Section 4 we will show how this specification can be correctly instrumented via program rewriting into a target language, hence we refer to the language introduced in this Section as our source language.

3.1. Source Language

Our source language model is essentially Featherweight Java (FJ) [21] with minor extensions including base types and an abstract notion of *library* methods for base types. The latter is important for an adequate consideration of taint propagation (e.g. on strings) in our model. FJ is a functional core calculus that includes class hierarchy definitions, subtyping, dynamic dispatch, and other basic features of Java. An FJ program is an expression `e` which is executed given a static *class table CT* which maintains class definitions. To describe program execution we will define a small step operational semantics relation on expressions `e` which we will take as synonymous with configurations as defined previously.

3.1.1. Syntax

The syntax of FJ is defined in Figure 1. We let `A`, `B`, `C`, `D` range over class names, `x` range over variables, `f` range over field names, and `m` range over method names. *Values*, denoted `v` or `u`, are objects, i.e. expressions of the form `new C`$(\mathtt{v}_1, \ldots, \mathtt{v}_n)$. We assume given an `Object` value that has no fields or methods. In addition to the standard expressions of FJ, we introduce a new form `C.m(e)`. This form is used to identify the method `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.

Conditional expressions are an important feature of the language for this presentation, since they are a control flow operation that should not be considered in a direct flow analysis. We assume that in any program setting true and false values, denote **T** and **F**, will be specified. When we consider base values

$\mathtt{L} ::= \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$	*classdefinitions*
$\mathtt{K} ::= \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}$	*constructors*
$\mathtt{M} ::= \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$	*methods*
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \mid \mathtt{if}\ \mathtt{e}\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e} \mid \mathtt{C.m}(\mathtt{e})$	*expressions*
$\mathtt{E} ::= [\,] \mid \mathtt{E.f} \mid \mathtt{E.m}(\overline{\mathtt{e}}) \mid \mathtt{v.m}(\overline{\mathtt{v}}, \mathtt{E}, \overline{\mathtt{e}}') \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}, \mathtt{E}, \overline{\mathtt{e}}') \mid \mathtt{if}\ \mathtt{E}\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e} \mid \mathtt{C.m}(\mathtt{E})$	*evaluation contexts*

Fig. 1. FJ Syntax

and library methods below in Section 3.1.6, we will define a particular boolean value that we will use in this presentation.

For brevity in this syntax, we use vector notations. Specifically we write $\overline{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$, similarly for $\overline{\mathtt{C}}$, $\overline{\mathtt{m}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{e}}$, etc., and we write $\overline{\mathtt{M}}$ 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 $\overline{\mathtt{m}}$, we write $\mathtt{m} \in \overline{\mathtt{m}}$. Vector notation is also used to abbreviate sequences of declarations; we let $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $\overline{\mathtt{C}}\ \overline{\mathtt{f}};$ 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 $\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};$ 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.

3.1.2. 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. Given a class table CT, the definitions of $mbody_{CT}(\mathtt{m}, \mathtt{C})$ and $fields_{CT}(\mathtt{C})$ are given in Figure 2.

3.1.3. Method type lookup

Just as we've defined a function for looking up method bodies in the class table, we also define a function $mtype_{CT}(\mathtt{C}, \mathtt{m})$ that will look up types of a method $\mathtt{C.m}$ in a class table in Figure 2. Although we omit FJ type analysis from this presentation, method type lookup will be useful for taint analysis instrumentation (Definition 4.1).

3.1.4. Operational semantics

Now, we can define the operational semantics of FJ. The reduction relation is binary, of the form $\kappa \rightarrow \kappa'$, and is defined via the inference rules in Figure 3.

The definition of $\rightarrow$ assumes given a class table CT which is typically clear from context, but we will write $CT \vdash \kappa \rightarrow \kappa'$ to disambiguate class tables used in reductions when necessary. The definition also assumes that boolean values $\mathbf{T}$ and $\mathbf{F}$ are specified. We use $\rightarrow^*$ to denote the reflexive, transitive closure of $\rightarrow$, and we use $\rightarrow^n$ to denote an n-step reduction. We will also use the notion of an *execution trace* τ to range over sequences of configurations $\kappa_0 \ldots \kappa_n$ where $\kappa_i \rightarrow \kappa_{i+1}$ for all $0 \leqslant i < n$. Note that an execution trace τ may represent the partial execution of a program, i.e. the trace τ may be extended with additional configurations as the program continues execution. In general we will write $CT \vdash_{\rightarrow} \tau$ to disambiguate the class table CT and reduction relation $\rightarrow$ used for a trace τ when it is not clear from context.

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

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

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

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

Fig. 2. Object Field, Method Body, and Method Type Lookup

Context
$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{E[e]} \to \texttt{E[e}'\texttt{]}}$$

Field
$$\frac{\mathit{fields}_{CT}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}} \qquad \texttt{f}_i \in \overline{\texttt{f}}}{\texttt{new C}(\overline{\texttt{v}}).\texttt{f}_i \to \texttt{v}_i}$$

Invoke
$$\frac{\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}, \texttt{e}}{\texttt{new C}(\overline{\texttt{v}}).\texttt{m}(\overline{\texttt{u}}) \to \texttt{C.m}(\texttt{e}[\texttt{new C}(\overline{\texttt{v}})/\texttt{this}][\overline{\texttt{u}}/\overline{\texttt{x}}])}$$

IfT
$\texttt{if}\ \mathbf{T}\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2 \to \texttt{e}_1$

IfF
$\texttt{if}\ \mathbf{F}\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2 \to \texttt{e}_2$

Return
$\texttt{C.m(v)} \to \texttt{v}$

Fig. 3. Operational Semantics for FJ

3.1.5. Top-Level Programs

We define *top-level programs* $\mathfrak{p}(\mathbf{a})$ as programs of the form:

$$\texttt{new TopLevel().main}(\mathbf{a})$$

where $\mathbf{a}$ is a primitive object $\texttt{new C}(\overline{v})$. We assume that all class tables CT include an entry point `TopLevel.main` with formal parameter `attack`, where `TopLevel` objects have no fields. We write $\mathfrak{p}(\mathbf{a}) \Downarrow \tau$ iff trace τ begins with the configuration $\mathfrak{p}(\mathbf{a})$.

3.1.6. Library Methods

In order to study dynamic integrity taint analysis in FJ, we extend the 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 [22] which have not been considered in previous formal work on taint analysis [1]. Since static analysis is not a topic of this paper, for brevity we omit the standard FJ type analysis which is described in [21].

The abstract calculus described above is not particularly interesting with respect to direct information flow and integrity propagation, especially since method dispatch and conditional expressions are control flows that are discounted in direct data flow. More interesting is the manner in which taint propagates through base values and library operations, since direct flows 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 [23], 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 [23]. Clearly this strategy introduces a potential for over-taint. Other systems do not propagate taint from strings to their component characters when decomposed [22], 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 base values and their manipulation. Let a *primitive field* be a field containing a base value. We call a *base type* any class with primitive fields only, and a *library method* is any method that operates on base type objects, defined in a primitive class. We expect primitive objects to be object wrappers for primitive values (e.g., `Int(5)` wrapping primitive value `5`), and library methods to be object-oriented wrappers over primitive operations (e.g., `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 *LibMeths* be the set of library method names paired with their corresponding base classes in *BaseTypes*.

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

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

The body of any library method is required to be of the form $\mathtt{return\ new\ C}(\overline{\mathtt{e}}_1, \ldots, \overline{\mathtt{e}}_n)$ where `C` is a primitive class.

We define the meaning of operations *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 follows:

$$+(n_1, n_2) \approx n_1 + n_2$$

Then we can add to the definition of `Int` in *CT* a method `Plus` to support arithmetic in programs:

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

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:

$$\texttt{String concat(String x)}\ \{\ \texttt{return(new String(@(this.val,x.val)));}\ \}$$

A boolean class `Bool` can be defined on the basis of constants *true* and *false* and standard boolean connectives– we will subsequently use this encoding for values **T** and **F** and conditional guards:

$$b \in \{\mathit{true}, \mathit{false}\} \qquad \mathbf{T} \triangleq \texttt{new Bool}(\mathit{true}) \qquad \mathbf{F} \triangleq \texttt{new Bool}(\mathit{false}) \qquad \wedge(b_1, b_2) \approx b_1 \wedge b_2$$

$$\vee(b_1, b_2) \approx b_1 \vee b_2 \qquad\qquad \neg(b) \approx \neg b$$

These boolean values can represent the results of base object comparison operators such as a string equality test:

$$eq(s_1, s_2) \approx b = \begin{cases} \mathit{true} \text{ if } s_1 = s_2 \\ \mathit{false} \text{ otherwise} \end{cases}$$

$$\texttt{String eq(String x)}\ \{\ \texttt{return(new Bool(}eq\texttt{(this.val,x.val)));}\ \}$$

3.2. In-Depth Integrity Analysis Specified Logically

In this section, we demonstrate how in-depth integrity taint analysis for FJ can be expressed as a single uniform policy separate from code. To accomplish this we interpret program traces as information represented by a logical fact base in the style of Datalog. We then define a predicate called Shadow that inductively constructs a "shadow" of configurations reflecting the taint of values.

Java-based taint analyses naturally tend to be object-based, i.e. low-integrity values are objects conceptually, and objects have an assigned taint level in the implementation. The types of tainted objects vary depending on the analysis, but most emphasize taint of base values. We will likewise focus on taint of base values, though we will support taint labeling of *all* objects. This is partly to generalize the representation, but also for formal convenience– In our logical specification of taint analysis, a shadow expression has a syntactic structure that matches the configuration expression, and associates integrity levels (including "high" $\circ$ and "low" $\bullet$) with particular objects via shape conformance.

Example 3.1. *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:*

$$\texttt{new C().m(new String(}s_1\texttt{), new String(}s_2\texttt{))}$$

Its proper shadow is:

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

$$\begin{aligned}
&\mathit{toFOL}(\mathtt{v}, n) = \{\text{Value}(n, \mathtt{v})\}\\
&\mathit{toFOL}(\mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f}], n) = \{\text{GetField}(n, \mathtt{new\ C}(\overline{\mathtt{v}}), \mathtt{f}), \text{Context}(n, \mathtt{E})\}\\
&\mathit{toFOL}(\mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}})], n) = \{\text{Call}(n, \mathtt{C}, \overline{\mathtt{v}}, \mathtt{m}, \overline{\mathtt{u}}), \text{Context}(n, \mathtt{E})\}\\
&\mathit{toFOL}(\mathtt{E}[\mathtt{C.m}(\mathtt{v})], n) = \{\text{ReturnValue}(n, \mathtt{C}, \mathtt{m}, \mathtt{v}), \text{Context}(n, \mathtt{E})\}\\
&\mathit{toFOL}(\mathtt{E}[\mathit{Op}(\overline{\mathtt{v}})], n) = \{\text{PrimCall}(n, \mathit{Op}, \overline{\mathtt{v}}), \text{Context}(n, \mathtt{E})\}\\
&\mathit{toFOL}(\mathtt{E}[\text{if } \mathbf{T} \text{ then } \mathtt{e}_1 \text{ else } \mathtt{e}_2], n) = \{\text{IfT}(n, \mathtt{e}_1, \mathtt{e}_2), \text{Context}(n, \mathtt{E})\}.\\
&\mathit{toFOL}(\mathtt{E}[\text{if } \mathbf{F} \text{ then } \mathtt{e}_1 \text{ else } \mathtt{e}_2], n) = \{\text{IfF}(n, \mathtt{e}_1, \mathtt{e}_2), \text{Context}(n, \mathtt{E})\}.
\end{aligned}$$

Fig. 4. Interpreting Expressions as Formulas via *toFOL*$(\cdot)$.

On the basis of shadow expressions that correctly track integrity, we can logically specify prospective taint analysis as a property of shadowed trace information, and retrospective taint analysis as a function of shadowed trace information. An extended example of a shadowed trace is presented in Section 3.2.4.

3.2.1. Taint Tracking as a Logical Trace Property

In order to specify taint tracking, we define the mapping *toFOL*$(\cdot)$ that shows how we concretely model execution traces in FOL. We develop *toFOL*$(\cdot)$ that interprets FJ traces as sets of logical facts (a fact base). Intuitively, in the interpretation each configuration is represented by a Context predicate representing the evaluation context, and a predicate representing the redex (e.g. Call). Each of these predicates has an initial natural number argument denoting a "timestamp" that orders configurations in a trace.

Definition 3.1. *We define toFOL*$(\cdot)$ *as a mapping on traces and configurations:*

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

such that $\mathit{toFOL}(\sigma) = \bigcup_i \mathit{toFOL}(\kappa_i, i)$ *for* $\sigma = \kappa_1 \cdots \kappa_k$. *We define* $\mathit{toFOL}(\kappa, n)$ *in Figure 4.*

Integrity Identifiers. We introduce an integrity identifier t that denotes the integrity level associated with objects. To support a notion of "partial endorsement" for partially trusted sanitizers, we define three taint labels, to denote high integrity ($\circ$), low integrity ($\bullet$), and uncertain integrity ($\odot$). We refer to these levels as tainted, untainted, and maybe tainted, respectively.

$$t ::= \circ \mid \odot \mid \bullet$$

We specify an ordering $\leqslant$ on these labels denoting their integrity relation:

$$\bullet \leqslant \odot \leqslant \circ$$

For simplicity in this presentation we will assume that all *Sanitizers* are partially trusted and cannot raise the integrity of a tainted or maybe tainted object beyond maybe tainted. It would be possible to include both trusted and untrusted sanitizers without changing the formalism.

We posit the usual meet $\wedge$ and join $\vee$ operations on taint lattice elements, and introduce logical predicates meet and 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.

3.2.2. Shadow Traces, Taint Propagation, and Sanitization

Shadow traces reflect taint information of objects as they are passed around programs. Shadow traces are comprised of shadow expressions and contexts which are terms in the logic with the following syntax. Note the structural conformance with closed $\mathtt{e}$ and $\mathtt{E}$, but with primitive values replaced with a single dummy value δ 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 values in objects– structural conformance is necessary since multiple values can occur in the same structured expression, and labels in shadows need to line up with their corresponding values in expressions. This is illustrated above in Example 3.1, and is discussed at more length in Section 1.3 in an example that we flesh out here in Section 3.2.4.

$$\begin{aligned}
sv &::= \mathtt{shadow}\ \mathtt{C}(t, \overline{sv}) \mid \delta \\
se &::= sv \mid se.\mathtt{f} \mid se.\mathtt{m}(\overline{se}) \mid \mathtt{shadow}\ \mathtt{C}(t, \overline{se}) \mid \mathtt{C.m}(se) \mid Op(\overline{se}) \mid \mathtt{if}\ se\ \mathtt{then}\ se\ \mathtt{else}\ se \\
SE &::= [\,] \mid SE.\mathtt{f} \mid SE.\mathtt{m}(\overline{se}) \mid sv.\mathtt{m}(\overline{sv}, SE, \overline{se}') \mid \mathtt{shadow}\ \mathtt{C}(t, \overline{sv}, SE, \overline{se}') \mid \mathtt{C.m}(SE) \mid \\
&\qquad Op(\overline{sv}, SE, \overline{se}) \mid \mathtt{if}\ SE\ \mathtt{then}\ se\ \mathtt{else}\ se
\end{aligned}$$

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 3.2. *Shadow method bodies are defined by the function smbody.*

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

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

$$\begin{aligned}
&srewrite(\mathtt{x}) = \mathtt{x} \\
&srewrite(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})) = \mathtt{shadow}\ \mathtt{C}(\circ, srewrite(\overline{\mathtt{e}})) \\
&srewrite(\mathtt{e.f}) = srewrite(\mathtt{e}).\mathtt{f} \\
&srewrite(\mathtt{e.m}(\overline{\mathtt{e}}')) = srewrite(\mathtt{e}).\mathtt{m}(srewrite(\overline{\mathtt{e}}')) \\
&srewrite(\mathtt{C.m}(\mathtt{e})) = \mathtt{C.m}(srewrite(\mathtt{e})) \\
&srewrite(Op(\overline{\mathtt{e}})) = Op(srewrite(\overline{\mathtt{e}})) \\
&srewrite(\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3) = \mathtt{if}\ srewrite(\mathtt{e}_1)\ \mathtt{then}\ srewrite(\mathtt{e}_2)\ \mathtt{else}\ srewrite(\mathtt{e}_3) \\
&srewrite(v) = \delta
\end{aligned}$$

The predicate Match defined in Figure 5 allows deconstruction of a shadow expression se into its constituent shadow context SE and shadow expression se' in the hole– that is, if $\mathrm{Match}(se, SE, se')$ then $se = SE[se']$.

$$\begin{array}{l}
\text{Match}(sv, [\,], sv).\\
\text{Match}(\texttt{shadow C}(t,\overline{sv}).\texttt{f}_i, [\,], \texttt{shadow C}(t,\overline{sv}).\texttt{f}_i).\\
\text{Match}(\texttt{shadow C}(t,\overline{sv}).\texttt{m}(\overline{su}), [\,], \texttt{shadow C}(t,\overline{sv}).\texttt{m}(\overline{su})).\\
\text{Match}(\texttt{C.m}(sv), [\,], \texttt{C.m}(sv)).\\
\text{Match}(\texttt{if } sv \texttt{ then } se_1 \texttt{ else } se_2, [\,], \texttt{if } sv \texttt{ then } se_1 \texttt{ else } se_2).\\
\text{Match}(se, SE, se') \implies \text{Match}(se.\texttt{f}, SE.\texttt{f}, se').\\
\text{Match}(se, SE, se') \implies \text{Match}(se.\texttt{m}(\overline{se}), SE.\texttt{m}(\overline{se}), se').\\
\text{Match}(se, SE, se') \implies \text{Match}(sv.\texttt{m}(\overline{sv}, se, \overline{se}), sv.\texttt{m}(\overline{sv}, SE, \overline{se}), se').\\
\text{Match}(se, SE, se') \implies \text{Match}(\texttt{shadow C}(t,\overline{sv}, se, \overline{se}), \texttt{shadow C}(t,\overline{sv}, SE, \overline{se}), se').\\
\text{Match}(se, SE, se') \implies \text{Match}(\texttt{C.m}(se), \texttt{C.m}(SE), se').\\
\text{Match}(se, SE, se') \implies \text{Match}(Op(\overline{sv}, se, \overline{se}), Op(\overline{sv}, SE, \overline{se}), se').\\
\text{Match}(se, SE, se') \implies \text{Match}(\texttt{if } se \texttt{ then } se_1 \texttt{ else } se_2, \texttt{if } SE \texttt{ then } se_1 \texttt{ else } se_2, se')
\end{array}$$

Fig. 5. Match Predicate Definition.

Next, in Figure 6, we define a predicate Shadow(n, se) where se is the relevant shadow expression at execution step n, establishing an ordering for the shadow trace. 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 Ops 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 Shadow inductively shadows the full trace. 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 Match predicate definition.[3]

With respect to control flow, the most notable rules of Shadow are those governing conditional branching, which ignore the taint of the guard, and method dispatch, which ignore the taint of the object associated with the dispatched method. Since we focus on base value taint, method dispatch is essentially a non-issue, however conditional branching is directly dependent on base values so ignoring the taint of the guard explicitly ignores indirect data flow.

Taint Propagation and Endorsement. The propagation of taint in the model described in Section 1.2 is embedded in the definition of Shadow, in particular we assume a set of *Sanitizers*. For elements of *Sanitizers*, if the input is tainted then the result is considered to be only partially endorsed (maybe tainted). For library methods, taint is propagated given a user-defined predicate Prop(t, ι) where ι 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:

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

[3]Some notational liberties are taken in Figure 6 regarding expression and context substitutions, which are defined using predicates elided for brevity.

Later in Section 5.2.1 we will discuss formal semantic conditions on library methods that ensure sound taint propagation.

3.2.3. In-Depth Integrity Taint Analysis Policies

Now we can logically specify an in-depth policy for integrity taint analysis, as proposed originally in Section 1.2. In particular we assume a set *Sanitizers* and a set *SSOs*. Since objects may inherit a sanitizer or SSO from a superclass, we require that *Sanitizers* and *SSOs* are closed under inheritance as a sanity condition, as follows:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathtt{D.m} \in \mathit{SSOs}}{\mathtt{C.m} \in \mathit{SSOs}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathtt{D.m} \in \mathit{Sanitizers}}{\mathtt{C.m} \in \mathit{Sanitizers}}$$

The in-depth policy has both prospective and retrospective component– the former is defined as a safety property [14], while the latter is defined as a logging specification. The prospective component of the policy must identify traces where a tainted value is passed to a secure method. To this end, in Figure 7 we define the predicate BAD which identifies traces that should be rejected as unsafe– a bad trace is any in which an SSO is executed with a tainted argument. The retrospective component specifies that data of questionable integrity that is passed to a secure method should be logged. The relevant logging specification is specified in terms of a predicate MaybeBAD also defined in Figure 7.

Definition 3.3. *Let X be the set of rules in Figures 5, 6, and 7 and the set of user-defined rules for* Prop. *The prospective integrity taint analysis policy is defined as the set of traces that are either free from or end in* BAD *configurations.* [4]

$$\mathrm{SP}_{\mathrm{taint}} = \{\tau\kappa \mid (\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\mathrm{BAD}\}} = C(\varnothing)\}.$$

The retrospective integrity taint analysis policy is the following logging specification, that uses spec as introduced in Definition 2.13. This logging policy maps traces to the set of maybe tainted objects that enter SSOs.

$$LS_{\mathrm{taint}} = spec(X, \mathrm{MaybeBAD})$$

We immediately observe that $\mathrm{SP}_{\mathrm{taint}}$ is a safety property:

Lemma 3.1. $\mathrm{SP}_{\mathrm{taint}}$ *is a safety property.*

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

Definition 3.4. *We call a program* $\mathtt{p}(\mathbf{a})$ safe *iff for all* τ *it is the case that* $\mathtt{p}(\mathbf{a}) \Downarrow \tau$ *implies* $\tau \in \mathrm{SP}_{\mathrm{taint}}$. *We call the program* unsafe *iff there exists some trace* τ *such that* $\mathtt{p}(\mathbf{a}) \Downarrow \tau$ *and* $\tau \notin \mathrm{SP}_{\mathrm{taint}}$.

[4]This latter condition is not necessary for the specification, and may seem extraneous, but it is in place to allow a clean proof correspondence with the implementation as detailed in Section 4.2. In the implementation, taint checks occur one execution step *after* an SSO is called, which in the case of an unsafe call will thus occur one step after the BAD configuration, but before anything bad actually happens, which blocking checks prevent.

$$\mathrm{Shadow}(1, \texttt{shadow TopLevel}(\circ).\texttt{main}(\texttt{shadow C}(\bullet, \overline{\delta}))).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, sv.\texttt{m}(\overline{sv}')) \wedge \texttt{C.m} \notin \mathit{LibMeths} \wedge \mathit{smbody}_{CT}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.se' \implies$$
$$\mathrm{Shadow}(n+1, SE[\texttt{C.m}(se'[\overline{sv}'/\overline{x}][sv/\texttt{this}])]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{shadow C}(t_0, \overline{sv}).\texttt{m}(\overline{\texttt{shadow C}(t, \overline{sv})})) \wedge \texttt{C.m} \in \mathit{LibMeths} \wedge \mathit{smbody}_{CT}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{shadow D}(\circ, \overline{se}) \wedge$$
$$\mathrm{Prop}(t, \texttt{C.m}(t_0, \overline{t})) \implies \mathrm{Shadow}(n+1, SE[\texttt{C.m}(\texttt{shadow D}(t, \overline{se})[\texttt{shadow C}(t_0, \overline{sv})/\texttt{this}][\overline{\texttt{shadow C}(t, \overline{sv})})/\overline{\texttt{x}}])]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{shadow C}(t, \overline{sv}).\texttt{f}_\texttt{i}) \implies \mathrm{Shadow}(n+1, SE[sv_i]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, Op(\overline{\delta})) \implies \mathrm{Shadow}(n, SE[\delta]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{C.m}(\texttt{shadow D}(t, \overline{sv}))) \wedge \texttt{C.m} \in \mathit{Sanitizers} \implies \mathrm{Shadow}(n+1, SE[\texttt{shadow D}(t \vee \odot, \overline{sv})]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{C.m}(sv)) \wedge \texttt{C.m} \notin \mathit{Sanitizers} \implies \mathrm{Shadow}(n+1, SE[sv]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{if } sv \texttt{ then } se_1 \texttt{ else } se_2) \wedge \mathrm{IfT}(n, \texttt{e}_1, \texttt{e}_2) \implies \mathrm{Shadow}(n+1, SE[se_1]).$$

$$\mathrm{Shadow}(n, se) \wedge \mathrm{Match}(se, SE, \texttt{if } sv \texttt{ then } se_1 \texttt{ else } se_2) \wedge \mathrm{IfF}(n, \texttt{e}_1, \texttt{e}_2) \implies \mathrm{Shadow}(n+1, SE[se_2])$$

Fig. 6. Shadow Predicate Definition.

$$\mathrm{Match}(se, SE, \texttt{shadow C}(t, \overline{sv}).\texttt{m}(\texttt{shadow D}(t', \overline{sv}'))) \wedge \mathrm{Shadow}(n, se) \wedge \mathrm{Call}(n, \texttt{C}, \overline{\texttt{v}}, \texttt{m}, \texttt{u}) \wedge \texttt{C.m} \in \mathit{SSOs} \implies \mathrm{SsoTaint}(n, t', \texttt{u}).$$

$$\mathrm{SsoTaint}(n, \bullet, \texttt{u}) \implies \mathrm{BAD}(n). \qquad \mathrm{SsoTaint}(n, t, \texttt{u}) \wedge t \leqslant \odot \implies \mathrm{MaybeBAD}(\texttt{u}).$$

Fig. 7. Predicates for Specifying Prospective and Retrospective Properties

3.2.4. Extended Example: Reduction and Shadowing

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 3.2. *Assume that sanitizer and SSO methods* `Sec.sanitize` *and* `Sec.secureMeth` *are identity functions for the sake of brevity, i.e.:*

$$\mathit{mbody}_{CT}(\texttt{Sec}, \texttt{sanitize}) = \texttt{x}, \texttt{x} \qquad \mathit{mbody}_{CT}(\texttt{Sec}, \texttt{secureMeth}) = \texttt{x}, \texttt{x}$$

and let $\mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel})$ *be:*

```
attack,
new Sec().secureMeth(new Sec().sanitize(attack.concat(new String("world")))
```

Assume also that an input string `"hello "` *is tainted with low integrity– Figure 8 depicts a source trace given the initial configuration:*

```
new TopLevel().main(new String("hello "))
```

$$\begin{gathered}
\mathtt{p}(\texttt{new String}(''\texttt{hello}''))\\
\rightarrow^{5}\\
\texttt{TopLevel.main}(\texttt{new Sec().secureMeth}(\texttt{new Sec().sanitize}(\texttt{new String}(''\texttt{hello world}''))))\\
\rightarrow^{2}\\
\texttt{TopLevel.main}(\texttt{new Sec().secureMeth}(\texttt{new String}(''\texttt{hello world}'')))\\
\rightarrow^{2}\\
\texttt{TopLevel.main}(\texttt{new String}(''\texttt{hello world}''))\\
\rightarrow\\
\texttt{new String}(''\texttt{hello world}'')
\end{gathered}$$

Fig. 8. Example 3.2: Source Trace.

$$\begin{array}{l}
\mathrm{Shadow}(1, \texttt{shadow TopLevel}(\circ)\texttt{.main}(\texttt{shadow String}(\bullet, \delta)))\\
\mathrm{Shadow}(5, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\texttt{shadow String}(\bullet, \delta)))))\\
\mathrm{Shadow}(7, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow String}(\odot, \delta))))\\
\mathrm{Shadow}(9, \texttt{TopLevel.main}(\texttt{shadow String}(\odot, \delta)))\\
\mathrm{Shadow}(10, \texttt{shadow String}(\odot, \delta))
\end{array}$$

Fig. 9. Example 3.2: Shadow Expressions.

with some reduction steps elided to highlight calls to `Sec.sanitize` *and* `Sec.secureMeth`*. In Figure 9 we show shadows of configurations highlighted (depicted) in the source trace. We note this trace is in* $\mathrm{SP}_{\mathrm{taint}}$ *and hence is safe.*

4. Correct 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 *Phos* that models the Phosphor rewriting algorithm, defined in Section 4.1. It adds taint label fields to all objects, and operations for appropriately propagating taint along direct flow paths. In addition to blocking behavior to enforce prospective checks, we incorporate logging instrumentation to support retrospective measures in the presence of partially trusted sanitization. We illustrate computation of instrumented code via an extended example in Section 4.1.4, which continues the (now running) example introduced in Section 3.2.4.

In Section 4.2 we follow the methods developed in Section 2 and show that *Phos* is semantics preserving, and that instrumented code generates sound and complete audit logs with respect to the logging specification LS_{taint} defined in Section 3.2.3. We will also show that instrumented code respects the safety property $\mathrm{SP}_{\mathrm{taint}}$ defined in the latter section.

4.1. In-Depth Taint Analysis Instrumentation

The target language $\mathrm{FJ}_{\mathrm{taint}}$ of the rewriting algorithm *Phos* has the same syntax as FJ except we add taint labels t as a form of primitive value v, 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 a convenient sequencing operation of the form `e;e` to target language expressions, and evaluation contexts of the form `E;e`.

4.1.1. The Phos Algorithm.

Now we define the program rewriting algorithm *Phos* as follows. It incorporates a rewriting function μ that assigns an untainted label to every object in an FJ source program. The class table is manipulated by *Phos* to specify a `taint` field for all objects, a `check` object method that blocks if the argument is tainted, and an `endorse` method for any object returned by a sanitizer.

Definition 4.1. *For any expression* `e`*, the expression* $\mu(\mathtt{e})$ *is syntactically equivalent to* `e` *except every subexpression* $\mathtt{new\ C}(\overline{\mathtt{e}})$ *is replaced with* $\mathtt{new\ C}(\circ, \overline{\mathtt{e}})$*. Given SSOs and Sanitizers, define:*

$$\mathit{Phos}(\mathtt{e}, CT) = (\mu(\mathtt{e}), \mathit{Phos}(CT))$$

where $\mathit{Phos}(CT)$ *is the smallest class table satisfying the axioms given in Figure 10. Furthermore, to correctly mark low integrity input as tainted, given class table CT and top-level program* $\mathfrak{p}(\mathbf{a})$ *where* $\mathbf{a} = \mathtt{new\ C}(\overline{v})$ *we define:*

$$\mathit{Phos}(\mathfrak{p}(\mathbf{a})) = \mathit{Phos}(\mathfrak{p}, CT)(\mathtt{new\ C}(\bullet, \overline{v}))$$

As discussed in Section 1, sanitization is typically taken to be "ideal" for integrity flow analyses, however in practice sanitization is imperfect, which creates an attack vector. To support retrospective measures specified in Definition 3.3, we define `endorse` so it takes object taint t to the join of t and $\odot$. The algorithm also adds a `log` method call to the beginning of *SSOs*, which will log objects that are maybe tainted or worse. The semantics of `log` are defined directly in the operational semantics of $\mathrm{FJ}_{\mathrm{taint}}$ below.

4.1.2. 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 [23]. We assume that library methods are instrumented to propagate taint as intended (i.e. in accordance with the user defined predicate Prop).

Here is how addition, string concatenation, and equality test, can be modified to propagate taint. Note the taint of arguments will be propagated to results by taking the meet of argument taint, thus reflecting the degree of integrity corruption:

```
Int plus(Int x)
    { return(new Int (∧(this.taint, x.taint), +(this.val, x.val))); }
```

$$\mathit{fields}_{\mathit{Phos}(CT)}(\texttt{Object}) = \texttt{Taint taint} \qquad \mathit{mbody}_{\mathit{Phos}(CT)}(\texttt{check}, \texttt{Object}) = \texttt{x}, \texttt{new Object(?x.taint)}; \texttt{x}$$

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

$$\frac{\texttt{C.m} \in \mathit{SSOs} \qquad \mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{e}}{\mathit{mbody}_{\mathit{Phos}(CT)}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{this.log(x)}; \texttt{this.check(x)}; \mu(\texttt{e})} \qquad \frac{\texttt{C.m} \in \mathit{Sanitizers} \qquad \mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}, \texttt{e}}{\mathit{mbody}_{\mathit{Phos}(CT)}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}, \mu(\texttt{e}).\texttt{endorse()}}$$

$$\frac{\texttt{C.m} \notin \mathit{Sanitizers} \cup \mathit{SSOs} \qquad \mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}, \texttt{e}}{\mathit{mbody}_{\mathit{Phos}(CT)}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}, \mu(\texttt{e})}$$

Fig. 10. Axioms for Rewriting Algorithm

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

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

4.1.3. Operational Semantics of FJ_{taint}

The operational semantics of FJ_{taint} are defined in Figure 11. Configurations in FJ are of the form $(\texttt{e}, \mathbb{L})$ where reductions are defined in terms of a labeled transition relation $\xrightarrow{\alpha}$ on configurations, where α is a possibly empty sequence ϵ of security events. These events are either *integrity* events $\mathit{iev}(\texttt{v})$ emitted when a check succeeds during evaluation as defined in the CheckPassed rule, or *endorsement* events $\mathit{eev}(\texttt{v})$, emitted when a value is endorsed as defined in the Endorsed rule. Labels are needed for our formulation of explicit integrity modulo endorsement, discussion in Section 5.

Audit logs $\mathbb{L}$ are added to configurations to support the retrospective security via audit logging, and are defined as sets of objects (values). The `log` method is the only one that interacts with the log in any way, and its semantics are specified in the Log and NoLog rules, where possibly tainted values are logged, and untainted ones are not. Note that we strip taint tags from values for logging– this is mainly to simplify the correspondence with LS_{taint} semantics for our technical development (where taint tags don't exist). We otherwise "inherit" the reduction semantics of FJ via the Reduce rule.

We write $\kappa_0 \xrightarrow[n]{\alpha_0 \cdots \alpha_{n-1}} \kappa_n$ iff $\kappa_i \xrightarrow{\alpha_i} \kappa_{i+1}$ for all $0 \leqslant i < n$, and write $\kappa \xrightarrow[*]{\alpha} \kappa'$ iff $\kappa \xrightarrow[n]{\alpha} \kappa'$ for some n. We may omit transition labels in cases where they are empty (ϵ) or not relevant to discussion, abusing notation $\to$, $\to^n$, and $\to^*$ as defined for FJ. We define traces as for FJ, and we write $\texttt{e} \Downarrow \tau$ iff τ begins with the configuration $(\texttt{e}, \emptyset)$.

4.1.4. Extended Example: Target Trace

Revisiting the example introduced in Section 3.2.4, we show execution of the rewritten program $\mathit{Phos}(\mathfrak{p}(\mathbf{a}))$ in Figure 12. By definition the rewritten top-level program is:

$$\texttt{new TopLevel}(\circ).\texttt{main}(\texttt{new String}(\bullet, ''\texttt{hello}''))$$

Reduce
$$\frac{\texttt{e} \rightarrow \texttt{e}' \qquad \neg(\texttt{e} = \texttt{E}[\texttt{C.m}(\texttt{v})] \wedge \texttt{m} \in \{\texttt{check}, \texttt{endorse}\})}{(\texttt{e}, \mathbb{L}) \xrightarrow{\epsilon} (\texttt{e}', \mathbb{L})}$$

CheckPassed
$$(\texttt{C.check}(\texttt{v}), \mathbb{L}) \xrightarrow{iev(\texttt{v})} (\texttt{v}, \mathbb{L})$$

Endorsed
$$(\texttt{C.endorse}(\texttt{v}), \mathbb{L}) \xrightarrow{eev(\texttt{v})} (\texttt{v}, \mathbb{L})$$

Sequence
$$(\texttt{v}; \texttt{e}, \mathbb{L}) \xrightarrow{\epsilon} (\texttt{e}, \mathbb{L})$$

Log
$$\frac{t \leqslant \odot}{(\texttt{u.log}(\texttt{new C}(t, \overline{\texttt{v}})), \mathbb{L}) \xrightarrow{\epsilon} (\texttt{new C}(t, \overline{\texttt{v}}), \mathbb{L} \cup \{\texttt{new C}(\overline{\texttt{v}})\})}$$

NoLog
$$\frac{t > \odot}{(\texttt{u.log}(\texttt{new C}(t, \overline{\texttt{v}})), \mathbb{L}) \xrightarrow{\epsilon} (\texttt{new C}(t, \overline{\texttt{v}}), \mathbb{L})}$$

Fig. 11. Operational Semantics of FJ_{taint}.

$$\Big(\textit{Phos}(\mathfrak{p}(\texttt{new String}(''\texttt{hello }''))), \emptyset\Big)$$
$$\xrightarrow[7]{\epsilon}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{new Sec}(\circ)\texttt{.secureMeth}(\texttt{new Sec}(\circ)\texttt{.sanitize}(\texttt{new String}(\bullet, ''\texttt{hello world}''))))\big), \emptyset\Big)$$
$$\xrightarrow{\epsilon}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{new Sec}(\circ)\texttt{.secureMeth}(\texttt{Sec.sanitize}(\texttt{new String}(\bullet, ''\texttt{hello world}'')\texttt{.endorse}())))\big), \emptyset\Big)$$
$$\xrightarrow[4]{eev(\texttt{new String}(\odot, ''\texttt{hello world}''))}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{new Sec}(\circ)\texttt{.secureMeth}(\texttt{new String}(\odot, ''\texttt{hello world}''))\big), \emptyset\Big)$$
$$\xrightarrow{\epsilon}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{Sec.secureMeth}(\texttt{new Sec}(\circ)\texttt{.log}(\texttt{v}_{\odot}); \texttt{new Sec}(\circ)\texttt{.check}(\texttt{v}_{\odot}); \texttt{v}_{\odot})\big), \emptyset\Big) \qquad \texttt{v}_{\odot} \triangleq \texttt{new String}(\odot, ''\texttt{hello world}'')$$
$$\xrightarrow[2]{\epsilon}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{Sec.secureMeth}(\texttt{new Sec}(\circ)\texttt{.check}(\texttt{v}_{\odot}); \texttt{v}_{\odot})\big), \{\texttt{new String}(''\texttt{hello world}'')\}\Big)$$
$$\xrightarrow[5]{iev(\texttt{new String}(\odot, ''\texttt{hello world}''))}$$
$$\Big(\texttt{TopLevel.main}\big(\texttt{v}_{\odot}\big), \{\texttt{new String}(''\texttt{hello world}'')\}\Big)$$
$$\xrightarrow{\epsilon}$$
$$\Big(\texttt{new String}(\odot, ''\texttt{hello world}''), \{\texttt{new String}(''\texttt{hello world}'')\}\Big)$$

Fig. 12. Example 3.2: Target Trace.

We note that additional reduction steps are necessary to evaluate instrumentation code in the target program, and that *eev* and *iev* events mark points during reduction when a value is endorsed and when it is checked.

4.2. Operational Properties of Phos

Now we can leverage machinery developed previously to demonstrate in-depth operational correctness of *Phos*, i.e. both prospective and retrospective operational correctness.

Recalling our definitions of semantics preservation, soundness, and completeness from Section 2, we state our main results as follows. These results tie together our relevant logging specification LS_{taint} and safety property SP_{taint} defined in Section 3.2.3. *We note that in this Section we will ignore transition labels α in target language reduction since they are irrelevant to the properties of interest, and will use $\rightarrow$ exclusively to refer to the reduction relation in FJ_{taint} defined in Section 4.*

Regarding our main result for retrospective security, we note that our definition is general with respect to *SSOs* and *Sanitizers* defined at the top-level, which fix LS_{taint} and SP_{taint}. Soundness and completeness as defined in Section 2.5 require definition of the notation $\tau \rightsquigarrow \mathbb{L}$, which for FJ_{taint} means that $\mathbb{L}$ is the log in the last configuration of τ. We define $\mathit{toFOL}(\mathbb{L}) = \{\text{MaybeBAD}(\texttt{v}) \mid \texttt{v} \in \mathbb{L}\}$, and thus $\lfloor \mathbb{L} \rfloor = C(\mathit{toFOL}(\mathbb{L}))$. Also as required, we need to define the relation $\cong$, establishing a semantic correspondence between FJ and FJ_{taint} traces. Intuitively, the relation holds on source, target trace pairs if the taint shadow of configurations in the source trace match up with the structure of configurations in the target trace modulo security instrumentation. This definition along with the detailed proofs of $\mathcal{R}$'s semantics preservation, soundness, and completeness (Theorem 4.1), and prospective correctness (Theorem 4.2) are given in the accompanying Technical Report [13].

Theorem 4.1. *For all $\mathfrak{p}(\mathbf{a})$, SSOs, and Sanitizers, let $\mathcal{R}(\mathfrak{p}(\mathbf{a}), LS_{\text{taint}}) = \mathit{Phos}(\mathfrak{p}(\mathbf{a}))$. Then $\mathcal{R}$ is semantics preserving, sound, and complete up to* SP_{taint}.

Since the safety property SP_{taint} has been defined for FJ, operational correctness for prospective security means that any rewritten unsafe programs are blocked by instrumentation. We can formalize this property as follows, noting it is a consequence of semantics preservation under $\cong$. This occurs because bad shadows in source code correspond to values that fail security checks in the target.

Definition 4.2. *An FJ_{taint} program* $\texttt{e}$ causes a security failure *iff*

$$(\texttt{e}, \emptyset) \rightarrow^* (\texttt{E}[\texttt{v.check(new C}(\bullet, \overline{\texttt{v}}))], \mathbb{L})$$

for some $\texttt{E}$, $\texttt{v}$, $\texttt{new C}(\bullet, \overline{\texttt{v}})$, *and* $\mathbb{L}$.

Operational correctness of the prospective component of *Phos* can then be stated as follows:

Theorem 4.2. *The FJ program $\mathfrak{p}(\mathbf{a})$ is unsafe iff $\mathit{Phos}(\mathfrak{p}(\mathbf{a}))$ causes a security failure.*

5. The Security Property of *Phos*

The semantics of information flow has been well studied and is typically characterized via noninterference properties, but surprisingly little work has been done to develop similar properties for taint analysis. In recent years it has been shown that direct flow of data confidentiality is not comparable with noninterference [10], i.e., there are both noninterfering programs with direct leakage of secret data to public domain, and programs without such direct leakages, but interfering. For instance consider the following two statements in a core imperative language, in which s and p are respectively secret and public variables:

$$\textbf{if } s = 0 \textbf{ then } p := s \textbf{ else } p := 0 \qquad\qquad \textbf{if } s = 0 \textbf{ then } p := 1 \textbf{ else } p := 2$$

The first statement is noninterfering, but direct flow of information from s to p exists, whereas the second statement is interfering due to the indirect flow from s to p, but there are no direct flows from s to p.

Formal definitions of taint analysis implementations do exist, but they are usually operational in nature. For example, in Section 4.2, we have established an operational correctness result for the prospective enforcement of direct integrity flow. In this section, we propose a hyperproperty to characterize the security property enforced by integrity taint analysis techniques. This hyperproperty is defined in a general, language-agnostic way, though in this Section we also show that the instrumentation of $\mathrm{FJ}_{\mathrm{taint}}$ programs by *Phos* enjoys this property as a correctness condition. We illustrate key points in Section 5.3.

5.1. Direct Integrity Flow Semantics: Explicit Integrity

We define *explicit integrity* as a semantic hyperproperty that builds on (dualizes) the notions of explicit secrecy [10] and attacker power [24]. Similar to explicit secrecy, explicit integrity is language-agnostic. In later sections, we discuss instantiation of this model for $\mathrm{FJ}_{\mathrm{taint}}$.

Intuitively, a program enjoys explicit secrecy if execution of its state transformation components does not affect the knowledge of a low confidentiality user. By formally specifying state transformation components, control flow operations (such as conditional expressions) can be omitted to only consider direct aka explicit program flows. *Knowledge* [25] is defined as the set of initial states configurable by a low confidentiality user that generate a particular sequence of observables– the smaller the set, the greater the knowledge. *Explicit knowledge* [10] restricts this concept to direct program data flow. In this section, we demonstrate how explicit knowledge can be "dualized" for direct integrity flow analysis and applied as a semantic framework for dynamic integrity taint analysis tools, particularly in functional languages with hierarchical data structures ($\mathrm{FJ}_{\mathrm{taint}}$).

Attacker power [24] is introduced as a counterpart to attacker knowledge in the context of integrity, as the set of low integrity inputs 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 [10] 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.

5.1.1. Model Specification

We formulate our explicit integrity semantics following [10]. We first define the interface for our framework. Let $\mathbf{K}$ be the set of program configurations for a given object language where κ ranges over configurations. Configurations consist of control and state segments. Control refers to code and state refers to data. Let $\mathbf{C}$ be the set of controls with $\mathbf{c}$ ranging over the elements of $\mathbf{C}$. Moreover, let $\mathbf{S}$ denote the set of states and $\mathbf{s}$ represent a given state. We also define a set of high integrity events, $\mathbf{E}$. A high integrity event $\mathbf{e}$ may refer to different computations in different language models and settings. For example, it could be as simple as assigning low integrity data to a high integrity variable, or invoking a method with low integrity data as its parameter to store that parameter in a database. We let α range over elements of $\mathbf{E}^*$. We assume the existence of the evaluation relation $\rightarrow \subseteq \mathbf{K} \times \mathbf{E}^* \times \mathbf{K}$ where $(\kappa, \alpha, \kappa') \in \rightarrow$

is denoted as $\kappa \xrightarrow{\alpha} \kappa'$. We use $\kappa \to \kappa'$ if α is empty (ϵ) or could be elided in the discussion. Notation $\to^*$ is used for reflexive and transitive closure of $\to$.

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 $\mathit{state} : \mathbf{K} \to \mathbf{S}$ and $\mathit{com} : \mathbf{K} \to \mathbf{C}$ extract the state and control segments of configurations, and $\langle \cdot, \cdot \rangle : \mathbf{C} \times \mathbf{S} \to \mathbf{K}$ construct a configuration from its control and state segments. These mappings need to satisfy the following property, for any κ:

$$\langle \mathit{com}(\kappa), \mathit{state}(\kappa) \rangle = \kappa.$$

We assume the existence of an entry point $[\cdot]$ in the controls denoted by $\mathbf{c}[\cdot]$ by which the attacker can inject low integrity input. The attacker input is denoted by $\mathbf{a}$. Then $\mathbf{c}[\mathbf{a}]$ represents a control in which the attacker has injected input $\mathbf{a}$. Note that an attack $\mathbf{a}$ is a data piece itself, i.e., $\mathbf{a}$ is a value.

We define extracted state transformers as follows. A consideration of state transformation, rather than complete program execution, allows us to focus only on direct program flow, rather than indirect control flow e.g. via conditionals. State transformers play the same role that explicit flow statements do in Weak Secrecy [26]. We note that this definition is a slight refinement of the analogous definition in [10]– in their work, a command is assumed to be compatible with all states, whereas we require compatibility of commands and states. This refinement is necessary due to structured expressions in HLLs such as Java, vs. lower level languages. However, we add a completeness condition expressed in Definition 5.3 that ensures we can compare all trust equivalent states via state transformation functions.

Definition 5.1. *Let $\kappa \to \kappa'$ and $\mathit{com}(\kappa) = \mathbf{c}$ for some $\mathbf{c}$. $f : \mathbf{S} \to \mathbf{S} \times \mathbf{E}^*$ is the function where $f(\mathbf{s}) = (\mathit{state}(\kappa''), \alpha)$ for all $\mathbf{s}$ such that $\langle \mathbf{c}, \mathbf{s} \rangle$ is defined and for the unique κ'' and α such that $\langle \mathbf{c}, \mathbf{s} \rangle \xrightarrow{\alpha} \kappa''$. We write $\kappa \to_f \kappa'$ to associate the state transformer f with the reduction $\kappa \to \kappa'$. This definition is then extended to multiple evaluation steps by composing state transformers at each step. Let $f(\mathbf{s}) = (\mathbf{s}', \alpha)$ and $g(\mathbf{s}') = (\mathbf{s}'', \alpha')$. Then, $(g * f)(\mathbf{s}) = (\mathbf{s}'', \alpha\alpha')$.*

We now define the power an attacker obtains by observing high integrity events. We capture this by defining a set of high integrity equivalent states that generate the same sequence of high integrity events. We posit the binary relation $=_\circ$ on $\mathbf{S}$ to denote high integrity equivalent (or trust equivalent) states. The general sense of this relation is that $\mathbf{s} =_\circ \mathbf{s}'$ if $\mathbf{s}$ and $\mathbf{s}'$ agree on high integrity data. The instantiation of the relation depends on the language model in which the states are defined. For a state $\mathbf{s}$ and some state transformer f, the state $\mathbf{s}'$ is considered as an element of the explicit attacker power if $\mathbf{s} =_\circ \mathbf{s}'$ and $\mathbf{s}'$ agrees with $\mathbf{s}$ on the generated high integrity events.

Definition 5.2. *We define* explicit attacker power *with respect to state $\mathbf{s}$ and state transformer f as follows, where projection on the ith element of a tuple is denoted by π_i.*

$$p_e(\mathbf{s}, f) = \{\mathbf{s}' \mid \mathbf{s} =_\circ \mathbf{s}', \pi_2(f(\mathbf{s})) = \pi_2(f(\mathbf{s}'))\}.$$

All state transformers must be complete in the following sense for this definition to be coherent:

Definition 5.3. *A state transformer f is* complete *iff for all $\mathbf{s}_1$, $\mathbf{s}_2$ where $\mathbf{s}_1 =_\circ \mathbf{s}_2$ we have $f(\mathbf{s}_1)$ is defined iff $f(\mathbf{s}_2)$ is defined.*

$$\frac{\textit{fields}_{Phos(CT)}(\mathtt{C}) = \overline{\mathtt{C}}\,\overline{\mathtt{f}} \qquad \mathtt{f}_i \in \overline{\mathtt{f}}}{\textit{select}_{\mathtt{f}_i}(\mathtt{E}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}).\mathtt{f}_i]) = (\mathtt{E}[\mathtt{v}_i], \epsilon)} \qquad \frac{\textit{mbody}_{Phos(CT)}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}, \mathtt{e} \qquad \mathtt{C.m} \notin \textit{LibMeths}}{\textit{call}_{\mathtt{C.m}}(\mathtt{E}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}})]) = (\mathtt{E}[\mathtt{C.m}(\mathtt{e}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})/\mathtt{this}][\overline{\mathtt{u}}/\overline{\mathtt{x}}])], \epsilon)}$$

$$\textit{return}(\mathtt{E}[\mathtt{C.m}(\mathtt{v})]) = (\mathtt{E}[\mathtt{v}], \epsilon) \qquad \frac{\mathtt{C.m} \in \textit{LibMeths} \qquad \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}_1).\mathtt{m}(\overline{\mathtt{u}}_1) \xrightarrow[*]{\epsilon} \mathtt{v}}{\textit{call}_{\mathtt{C.m}}(\mathtt{E}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}_1).\mathtt{m}(\overline{\mathtt{u}}_1)]) = (\mathtt{E}[\mathtt{v}], \epsilon)} \qquad \frac{\mathtt{u.check}(\mathtt{v}) \to^* \mathtt{v}}{\textit{check}(\mathtt{E}[\mathtt{u.check}(\mathtt{v})]) = (\mathtt{E}[\mathtt{v}], \textit{iev}(\mathtt{v}))}$$

$$\frac{\mathtt{u.endorse}(\mathtt{v}) \to^* \mathtt{v}'}{\textit{endorse}(\mathtt{E}[\mathtt{u.endorse}(\mathtt{v})]) = (\mathtt{E}[\mathtt{v}'], \textit{eev}(\mathtt{v}'))} \qquad \textit{if}_{\mathbf{T}}(\mathtt{E}[\mathtt{if}\ \mathtt{v}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2]) = (\mathtt{E}[\mathtt{e}_1], \epsilon)$$

$$\textit{if}_{\mathbf{F}}(\mathtt{E}[\mathtt{if}\ \mathtt{v}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2]) = (\mathtt{E}[\mathtt{e}_2], \epsilon) \qquad \textit{sequence}(\mathtt{E}[\mathtt{v}; \mathtt{e}]) = (\mathtt{E}[\mathtt{e}], \epsilon)$$

Fig. 13. Fundamental State Transformers Extracted from $\xrightarrow{\alpha}$.

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

Definition 5.4. *A control* **c** *satisfies* explicit integrity *for state* **s**, *iff* $\langle \mathbf{c}, \mathbf{s}\rangle \to_f^* \kappa'$ *implies that for any* $\mathbf{s}'$ *and* $\mathbf{s}''$, *if* $\mathbf{s}' =_\circ \mathbf{s}''$ *then we have* $\mathbf{s}'' \in p_e(\mathbf{s}', f)$. *A control* **c** *satisfies* explicit integrity *iff for any* **s**, **c** *satisfies explicit integrity for* **s**.

We can now consider explicit integrity in the presence of endorsement in the style of gradual release [25]. 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 5.5. *A control* **c** *satisfies* explicit integrity modulo endorsement *for state* **s** *iff* $\langle \mathbf{c}, \mathbf{s}\rangle \to_f^* \kappa' \xrightarrow{\alpha}_g^* \kappa''$ *and* $\alpha \notin \mathbf{E_{en}}^*$ *imply that* $p_e(\mathbf{s}, f) = p_e(\mathbf{s}, g * f)$.

5.2. An Instantiation with FJ_{taint}

In this section, we instantiate explicit integrity for FJ_{taint}. *Because audit logging and retrospective features are irrelevant to the technical development in this Section, we omit them and elide FJ_{taint} configurations to just expressions* `e`, *and take* `log` *to be the identity function.*

First, we define the required interface specified in Section 5.1, beginning with the definition of extracted state transformers for all features. These are extracted from the definition of $\to$– notably, the extracted state transformers for conditional expressions inline conditional branching, disregarding the actual **T** or **F** value of the guard, and eliminating the effects of indirect flow from state transformation functions.

Definition 5.6. *The state transformers for FJ_{taint} are composed of commands of the form select*$_{\mathtt{f}}$ *for all fields f (selection), call*$_{\mathtt{C.m}}$ *for all class, method pairs* `C.m` *(method dispatch), return (method return), endorse (endorsement), check (successful taint check within an SSO), sequence (sequencing), and if*$_{\mathbf{T}}$ *and if*$_{\mathbf{F}}$ *(branch inlining). The behavior of these fundamental extracted state transformers are defined in Figure 13.*

Our treatment of library methods, `check`, and `endorse` bear discussion since they consider these in an atomic, "big step" manner. As noted in Section 3.2.2, when taint is propagated by library methods,

$$\mathit{com}(\texttt{E[new C}(\overline{\texttt{v}})\texttt{.f]}) = \mathit{select}_{\texttt{f}} \qquad \mathit{com}(\texttt{E[new C}(\overline{\texttt{v}})\texttt{.m}(\overline{\texttt{u}})\texttt{]}) = \mathit{call}_{\texttt{C.m}} \quad \texttt{m} \notin \{\texttt{check}, \texttt{endorse}\}$$

$$\mathit{com}(\texttt{E[u.check(v)]}) = \mathit{check} \qquad \mathit{com}(\texttt{E[u.endorse(v)]}) = \mathit{endorse} \qquad \mathit{com}(\texttt{E[C.m(v)]}) = \mathit{return}$$

$$\mathit{com}(\texttt{E[if T then e}_1\texttt{ else e}_2\texttt{]}) = \mathit{if}_{\mathbf{T}} \qquad \mathit{com}(\texttt{E[if F then e}_1\texttt{ else e}_2\texttt{]}) = \mathit{if}_{\mathbf{F}}$$

Fig. 14. Definition of *com* for $\mathrm{FJ}_{\mathrm{taint}}$.

for efficiency or implementation convenience it may be the case that taint propagation is not correctly applied until computed results are returned. This includes `check` and `endorse`, since technically these are library methods as per the definition in Section 3.1.6. Thus we specify that the extracted state transformer of any library method treat it atomically with respect to internal computations. In addition to *check* and *endorse*, for library methods where no security related events can occur we define a class of state transformers $\mathit{call}_{\texttt{C.m}}$ for $\texttt{C.m} \in \mathit{LibMeths}$. This definition will also significantly simplify *our* proofs, and is irrelevant from a formal perspective since this definition yields the same observable events that a strict "small-step" definition of state transformers would for a given top-level program in the image of *Phos*.

Next, we define *com*, *state*, and $\langle\cdot,\cdot\rangle$ for $\mathrm{FJ}_{\mathrm{taint}}$. The command associated with a particular configuration `e` can be determined from its redex. We take the state of a configuration `e` to just be `e` itself, and combining a command and a state to obtain a configuration requires that the given command matches the form of the redex in the state– i.e. compatibility of the command and the state.

Definition 5.7. *We define* $\mathit{state}(\texttt{e}) = \texttt{e}$ *and define com as in Figure 14. We define* $\langle\cdot,\cdot\rangle$ *as in Figure 15.*

These definitions clearly satisfy the model requirements.

Lemma 5.1. *For any* $\mathit{FJ}_{\mathrm{taint}}$ *configuration* κ, *we have* $\langle \mathit{com}(\kappa), \mathit{state}(\kappa)\rangle = \kappa$.

Trust equivalence and state transformation. Now we define trust equivalence $=_\circ$ on $\mathrm{FJ}_{\mathrm{taint}}$ expressions as required. This definition requires structural conformance of related states (expressions), and requires agreement of base values except in the case of tainted base objects. Aside from satisfying the model definition, the definition of trust equivalence will be crucial in our proof of explicit integrity modulo endorsement, as it defines the necessary inductive invariant on extracted state transformations for this result.

Also, since endorsement may allow trust equivalent states to transform into non-structural equivalence, to satisfy the completeness requirement of Definition 5.3 we need to show that transformation preserves a weaker structural conformance relation $=_*$ on states (expressions). These relations are very similar with $=_*$ strictly weaker than $=_\circ$, and in proofs we will generally consider them together. Hence we define the metavariable $=_{\circledast}$ to range over $=_*$ and $=_\circ$.

Definition 5.8. *The* trust equivalence $=_\circ$ *and* shape conformance $=_*$ *relations on expressions are defined as the least relations inductively satisfying the rules in Figure 16, where* $=_{\circledast}$ *is a metavariable that ranges over* $=_\circ$ *and* $=_*$.

$$\langle \mathit{select}_{\mathtt{f}}, \mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f}]\rangle = \mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f}] \qquad \langle \mathit{call}_{\mathtt{C.m}}, \mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}})]\rangle = \mathtt{E}[\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}})] \qquad \langle \mathit{return}, \mathtt{E}[\mathtt{C.m}(\mathtt{v})]\rangle = \mathtt{E}[\mathtt{C.m}(\mathtt{v})]$$

$$\langle \mathit{check}, \mathtt{E}[\mathtt{u.check}(\mathtt{v})]\rangle = \mathtt{E}[\mathtt{u.check}(\mathtt{v})] \qquad \langle \mathit{endorse}, \mathtt{E}[\mathtt{u.endorse}(\mathtt{v})]\rangle = \mathtt{E}[\mathtt{u.endorse}(\mathtt{v})]$$

$$\langle \mathit{if}_{\mathbf{T}}, \mathtt{E}[\mathtt{if\ v\ then\ e_1\ else\ e_2}]\rangle = \mathtt{E}[\mathtt{if\ v\ then\ e_1\ else\ e_2}] \qquad \langle \mathit{if}_{\mathbf{F}}, \mathtt{E}[\mathtt{if\ v\ then\ e_1\ else\ e_2}]\rangle = \mathtt{E}[\mathtt{if\ v\ then\ e_1\ else\ e_2}]$$

Fig. 15. Definition of $\langle\cdot,\cdot\rangle$ for $\mathrm{FJ}_{\mathrm{taint}}$

$$\mathtt{x} =_{\circledast} \mathtt{x} \qquad v =_{\circ} v \qquad v_1 =_{*} v_2 \qquad \mathit{Op}(\bar{e}) =_{\circledast} \mathit{Op}(\bar{e}) \qquad \frac{\mathtt{e}_1 =_{\circledast} \mathtt{e}_2}{\mathtt{e}_1.\mathtt{f} =_{\circledast} \mathtt{e}_2.\mathtt{f}} \qquad \frac{\mathtt{e}_1, \overline{\mathtt{e}}_1 =_{\circledast} \mathtt{e}_2, \overline{\mathtt{e}}_2}{\mathtt{e}_1.\mathtt{m}(\overline{\mathtt{e}}_1) =_{\circledast} \mathtt{e}_2.\mathtt{m}(\overline{\mathtt{e}}_2)}$$

$$\frac{\mathtt{e}_1 =_{\circledast} \mathtt{e}_2}{\mathtt{C.m}(\mathtt{e}_1) =_{\circledast} \mathtt{C.m}(\mathtt{e}_2)} \qquad \frac{\overline{\mathtt{e}}_1 =_{\circledast} \overline{\mathtt{e}}_2}{\mathtt{new\ C}(t, \overline{\mathtt{e}}_1) =_{\circledast} \mathtt{new\ C}(t, \overline{\mathtt{e}}_2)} \qquad \frac{\mathtt{C} \in \mathit{BaseTypes}}{\mathtt{new\ C}(\bullet, v_1) =_{\circ} \mathtt{new\ C}(\bullet, v_2)} \qquad \frac{\mathtt{e}_1 =_{\circledast} \mathtt{e}_1' \qquad \mathtt{e}_2 =_{\circledast} \mathtt{e}_2'}{\mathtt{e}_1; \mathtt{e}_2 =_{\circledast} \mathtt{e}_1'; \mathtt{e}_2'}$$

$$\frac{\mathtt{e}_1^1 =_{\circledast} \mathtt{e}_1^2 \ \cdots\ \mathtt{e}_n^1 =_{\circledast} \mathtt{e}_n^2}{\mathtt{e}_1^1 \cdots \mathtt{e}_n^1 =_{\circledast} \mathtt{e}_1^2 \cdots \mathtt{e}_n^2} \qquad \frac{\mathtt{e}_1 =_{\circledast} \mathtt{e}_1' \qquad \mathtt{e}_2 =_{\circledast} \mathtt{e}_2' \qquad \mathtt{e}_3 =_{\circledast} \mathtt{e}_3'}{\mathtt{if\ e_1\ then\ e_2\ else\ e_3} =_{\circledast} \mathtt{if\ e_1'\ then\ e_2'\ else\ e_3'}}$$

Fig. 16. Definition of Trust Equivalence and Shape Conformance Relations on Expressions

5.2.1. Sanity Conditions on Library Methods

We define two sanity conditions for library methods: *not undertainting* and *not overtainting*. The former condition is required in the implementation in order to meet explicit integrity modulo endorsement, whereas the latter is a good practice in the implementation of taint analysis tools. Hereafter we will assume that library methods are not undertainting.

Definition 5.9. *We say* $\mathtt{C.m} \in \mathit{LibMeths}$ is not undertainting *iff for all* $\overline{\mathtt{v}}_1$*,* $\overline{\mathtt{u}}_1$*,* $\overline{\mathtt{v}}_2$*,* $\overline{\mathtt{u}}_2$ *where:*

$$\overline{\mathtt{v}}_1, \overline{\mathtt{u}}_1 =_{\circ} \overline{\mathtt{v}}_2, \overline{\mathtt{u}}_2 \qquad \mathit{call}_{\mathtt{C.m}}(\mathtt{new\ C}(\overline{\mathtt{v}}_1).\mathtt{m}(\overline{\mathtt{u}}_1)) = (\mathtt{v}_1, \epsilon) \qquad \mathit{call}_{\mathtt{C.m}}(\mathtt{new\ C}(\overline{\mathtt{v}}_2).\mathtt{m}(\overline{\mathtt{u}}_2)) = (\mathtt{v}_2, \epsilon)$$

we have $\mathtt{v}_1 =_{\circ} \mathtt{v}_2$.

For example, `String.concat` is not undertainting if the taint propagation policy is defined as in Section 3.2.2 where the taint of a concatenated string is the meet of its operands' taints, but it would be e.g. if its results were always untainted.

Not overtainting refines the precision of taint tracking with respect to a given state. Intuitively, a library method that only *directly* depends on its high integrity inputs is not overtainting if its results are untainted.

Definition 5.10. *We say* $\mathtt{C.m} \in \mathit{LibMeths}$ is not overtainting *with respect to input* $\overline{\mathtt{v}}_1, \overline{\mathtt{u}}_1$ *iff for all* $\overline{\mathtt{v}}_2$*,* $\overline{\mathtt{u}}_2$ *where:*

$$\overline{\mathtt{v}}_1, \overline{\mathtt{u}}_1 =_{\circ} \overline{\mathtt{v}}_2, \overline{\mathtt{u}}_2 \qquad \mathit{call}_{\mathtt{C.m}}(\mathtt{new\ C}(\overline{\mathtt{v}}_1).\mathtt{m}(\overline{\mathtt{u}}_1)) = (\mathtt{v}_1, \epsilon) \qquad \mathit{call}_{\mathtt{C.m}}(\mathtt{new\ C}(\overline{\mathtt{v}}_2).\mathtt{m}(\overline{\mathtt{u}}_2)) = (\mathtt{v}_2, \epsilon)$$

if $\mathtt{v}_1 = \mathtt{v}_2$ *then* $\mathtt{v}_1 = \mathtt{new\ D}(\circ, \overline{\mathtt{v}})$ *for some* $\overline{\mathtt{v}}$*.*

5.3. Extended Example

Assume given a class table *CT* containing sanitizer and SSO methods `Sec.sanitize` and `Sec.secureMeth`, which are identity functions for the sake of brevity, i.e.:

$$mbody_{CT}(\texttt{Sec}, \texttt{sanitize}) = \texttt{x}, \texttt{x} \qquad mbody_{CT}(\texttt{Sec}, \texttt{secureMeth}) = \texttt{x}, \texttt{x}$$

and let $mbody_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{attack}, \texttt{e}$ where `e` is:

```
if attack.eq(new String("foo")) then
    new Sec().secureMeth(attack)
else
    new Sec().secureMeth(new Sec().sanitize(attack))
```

Note that this is an example of a program that is unsafe by our definition, since a tainted value can flow directly into an SSO, though it is noninterfering modulo endorsement since that value can only be `new String("foo")` (similar to the example at the beginning of this Section). However, *Phos* will place a check that will ensure blocking of unsafe executions. We note that the execution of $Phos(\mathfrak{p}(\texttt{new String}(''\texttt{foo}'')))$ up to the point it gets stuck within `Sec.check` is associated with the following state transformer f:

$$f = sequence * return * call_{\texttt{Sec.log}} * call_{\texttt{Sec.secureMeth}} * if_{\mathbf{T}} * call_{\texttt{String.eq}}$$

Observe that $\pi_2(f(Phos(\mathfrak{p}(\mathbf{a})))) = \epsilon$ for any $\mathbf{a}$, trivially satisfying the requirements of explicit integrity modulo endorsement. Crucially, note that $\mathbf{a}$ need not be the string $''\texttt{foo}''$ in order for $f(Phos(\mathfrak{p}(\mathbf{a})))$ to be defined– even though the program $Phos(\mathfrak{p}(\mathbf{a}))$ would not take the **T** branch through the conditional during actual execution, it is "forced" that way by f. This is central to the definition of explicit attacker power with respect to $Phos(\mathfrak{p}(\texttt{new String}(''\texttt{foo}''))$ and f.

In contrast, the state transformer associated with the actual execution of $Phos(\mathfrak{p}(\texttt{new String}(s)))$ for $s \neq ''\texttt{foo}''$ up to the point it gets endorsed by `String.endorse` within `Sec.sanitize` is:

$$g = endorse * call_{\texttt{Sec.sanitize}} * if_{\mathbf{F}} * call_{\texttt{String.eq}}$$

We note that:

$$\pi_2(g(Phos(\mathfrak{p}(\texttt{new String}(s'))))) = eev(\texttt{new String}(s'))$$

for all s'. Furthermore, continued execution of $Phos(\mathfrak{p}(\texttt{new String}(s)))$ is associated with the following function h which takes the program through the successful check of the sanitized object:

$$h = check * sequence * return * call_{\texttt{Sec.log}} * call_{\texttt{Sec.secureMeth}}$$

We note that:

$$\pi_2(h * g(Phos(\mathfrak{p}(\texttt{new String}(s'))))) = eev(\texttt{new String}(s')), iev(\texttt{new String}(s'))$$

for all s'. Finally, we observe that:

$$p_e(Phos(\mathfrak{p}(\mathbf{a})), g) = p_e(Phos(\mathfrak{p}(\mathbf{a})), h * g) = \{Phos(\mathfrak{p}(\mathbf{a}))\}$$

for all **a**, satisfying the requirements of explicit integrity modulo endorsement.

Since f and $h * g$ represent all possible control flow paths through the program that can generate events, it is evident that $Phos(\mathfrak{p}(\mathbf{a}))$ satisfies explicit integrity modulo endorsement for all **a**.

5.4. Enforcement of Explicit Integrity Modulo Endorsement by Phos

Our general strategy is to show that non-endorsement events do not change attacker power as required by the definition of explicit integrity modulo endorsement. The complete proof details are given in our Technical Report [13].

Theorem 5.1. *If* $\mathtt{e}$ *is in the image of Phos, then it enjoys explicit integrity modulo endorsement.*

Proof. (Sketch) Proof by contradiction. If it does not enjoy explicit integrity modulo endorsement, then explicit attacker power is refined, i.e., different integrity events can be generated starting from trust equivalent states. This contradicts with an intermediary result reflecting on the preservation of integrity events by state transformers being applied on trust equivalent states. □

6. An Implementation of *Phos* in OpenMRS

In Section 1.1 we discussed an XSS vulnerability in the OpenMRS system (corrected in the current version) that inspired our interest in an in-depth taint analysis to better track data flow into secure operations and to enforce some level of sanitization. To explore and evaluate our proposed methods in practice, we have developed an automated analysis for OpenMRS by direct modification of the Phosphor system [5]. Our modification supports dynamic integrity taint analysis both prospectively and retrospectively. Our implementation is based on the formal model developed in previous sections, which enjoys a correctness guarantee. In this Section we describe our implementation and our evaluation of it.

6.1. Modifications to Phosphor

Out of the box, Phosphor provides a binary taint labeling scheme, with no support for endorsement. Users specify their security policy by identifying high integrity sinks, which are then automatically instrumented at the bytecode level with checks for low integrity inputs, by a combination of program rewriting and runtime mechanisms. Thus, to implement our in-depth taint analysis specification we needed to generalize the taint labeling scheme, add an endorsement mechanism, and add support for audit logging to the existing Phosphor codebase. This yielded our *Phos* implementation, as distinct from Phosphor.

Phosphor distinguishes only between two types of data– tainted and untainted. To support a generalized labeling scheme, in *Phos* we added to the Phosphor `Taint` class definition a field containing a `TaintLevel` enumeration. This latter type is endowed with a partial ordering that is specified by the programmer via an underlying graph definition, and join and meet operations. In our implementation

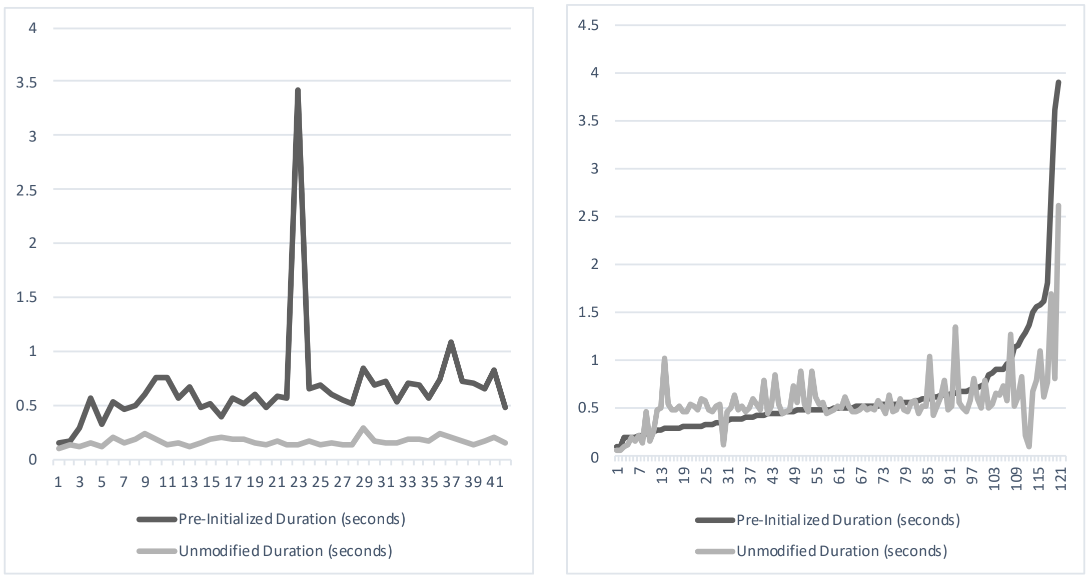

Fig. 17. *Phos* instrumentation timing overhead for OpenMRS for actions (left) and page loads (right). Numbers on the x-axis identify particular actions and page loads, and the y-axis is completion time in seconds.

we support the taint label lattice defined in Section 3.2 but this could be easily changed to accommodate others. We also define an `endorse` operation that takes the join of the input taint label and `MaybeTainted` as in this paper. Since Phosphor itself adds a `Taint` object to all program objects, these modifications are propagated through the system by the existing codebase.

As for ordinary Phosphor, in *Phos* we allow specification of secure sinks, however the rewriting algorithm adds instrumentation for audit logging of values at or below a specified taint level that reach any sink (`MaybeTainted` in our case). The following information is logged in such a case: the function name of any sink that had a tainted variable pass through it, the taint level of any sunken tainted variable, the value of sunken variables, and a stack trace of the thread when a tainted variable was sunk. Much of this information was already being collected in the unmodified Phosphor.

We also allow specification of a set of sanitizers in the same manner as secure sink specification, i.e. specific methods are identified in an initial configuration file provided when rewriting a program. These methods will have return values `endorse`d via insertion of that method. Thus the end product functions the same as the system specified in Section 4– an input set of *SSOs* and *Sanitizers* are provided, along with a program for instrumentation, and the program is rewritten with instrumentation to support $\mathrm{SP}_{\mathrm{taint}}$ and LS_{taint}. Our *Phos* implementation also supports a specification of low integrity sources at arbitrary taint levels. The implementation is available on a public GitHub [27], as well as our *Phos*-instrumented version of OpenMRS.

6.2. *OpenMRS Sources, Sinks, and Sanitizers*

To apply *Phos* to OpenMRS, it is necessary to identify sources, sinks, and sanitizers in the system. Since our concern is mainly defense against injection and XSS attacks, we focused on database interactions. OpenMRS in its current form uses the popular Hibernate ORM framework as a database API, which supports two ways of interacting with databases– via persistent relationally mapped object saving/loading, and via queries. We limited the scope of our work to focus on queries based on data in memory rather than persistent data, since the latter would require persistence of taint information and hence a far more complex implementation task.

The lists of sources, sinks, and sanitizers we identified in OpenMRS are provided in our implementation on GitHub [27]. Our method for identifying sinks and sanitizers was to leverage our knowledge of the Hibernate API. Specifically, to identify sinks, we searched the OpenMRS codebase for methods that employ Hibernate database write functionality. To identify sanitizers, we searched the OpenMRS codebase for methods that employ Hibernate sanitization functionality. The list of sources was determined by searching for methods that use `javax.servlet` functionality for recovering data from `POST` requests.

Another subtle but important detail of our integration of *Phos* with OpenMRS is that in OpenMRS, the arguments for the sinks are not necessarily tainted themselves, but rather are objects containing tainted member variables. Therefore, we also modified Phosphor to not only check sink arguments for a taint, but also argument member variables.

6.3. Implementation Evaluation

To evaluate our instrumentation of OpenMRS with *Phos*, we developed an automated testing method to evaluate correctness of the implementation, as well as timing and memory overhead. However, to understand our evaluation, certain details need to be explained.

Phosphor Initialization Overhead. When instrumenting Java classes with Phosphor, one can either run the software manually, specifying all of the source files to the program, or Phosphor can automatically detect and instrument uninstrumented code as the program runs. The latter option was chosen for this project due to the nature of OpenMRS and the onerous overhead of manual instrumentation.

As a consequence of instrumenting Java classes dynamically, an instrumentation overhead occurs as new uninstrumented source code is discovered. Thus, an initial run through a Phosphor-modified OpenMRS would be slower than consecutive runs, which was indeed observed by testing– the initial run of a particular method typically took about twice as long as subsequent runs.

Since our main concern in evaluation was to compare the overhead of instrumented vs. unmodified OpenMRS, and initialization overhead is arguably amortized to insignificance over long use sessions, the results we report here are only for pre-initialized testing runs. However we do note this additional overhead with the use of Phosphor's dynamic instrumentation feature.

Actions and Page Loads. The OpenMRS system has a web-based user interface, so we can partition its functionality into two major categories. The first is called *action* functionality, which results from submitting a form. Since form submission introduces tainted data that is potentially sanitized and destined for a secure sink, we can expect actions to incur overhead in instrumented OpenMRS and so are clearly important to consider.

OpenMRS also offers the potential for *page loads*, where users navigate between pages in the system by clicking links (but not submitting data). Since the underlying code is also instrumented in these situations, and continues to track taint, some overhead can also be observed. Thus we also evaluated overhead associated with page loads.

6.3.1. Experiments and Results

To evaluate *Phos*-instrumented OpenMRS, we developed a script that iterated over 42 actions, and over 121 page loads, recording timing and memory use, that we call a *test run*. We did a test run over

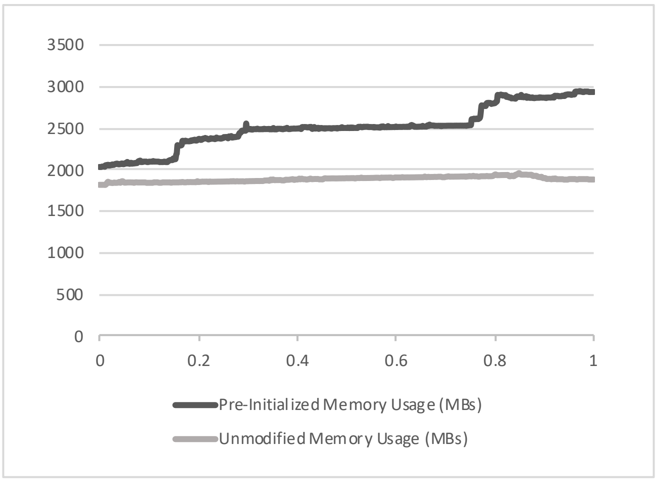

Fig. 18. Phosphor instrumentation memory overhead. The x-axis denotes the fraction completion of a test run, and the y-axis denotes memory used in MB.

unmodified OpenMRS to establish a baseline, and also did a test run over OpenMRS instrumented with our implementation of *Phos*. Finally, to evaluate how much our modifications impact Phosphor overhead, we did an actions-only test run over OpenMRS instrumented with pre-initialized unmodified Phosphor.

An initial concern of our evaluation was determining whether the system worked correctly, and whether data reaching sinks was maybe tainted, indicating sanitization, as well as being logged properly. We confirmed this, and did not discover instances of unsanitized data reaching sinks. Subsequently, we considered timing and memory consumption.

Timing. Our timing results are summarized in Table 1. Here we show the average time to complete each action and page load for the unmodified OpenMRS baseline, as well as OpenMRS instrumented with our implementation of *Phos*, and OpenMRS instrumented with pre-initialized unmodified Phosphor. These results demonstrate that instrumentation imposes a bit less than 3x overhead, while average times for completion are not onerous. Furthermore, our comparison of *Phos* and Phosphor shows that our modifications to did not add significant overhead to the taint analysis.

Table 1
Average timing and overhead for unmodified OpenMRS (baseline), versus instrumented with Phosphor and with *Phos*.

	Actions		**Loads**	
	Avg (secs)	Overhead	Avg (secs)	Overhead
OpenMRS Baseline	.236	-	.567	-
OpenMRS + Phosphor	.614	261%	-	-
OpenMRS + *Phos*	.670	284%	.636	112%

Figure 17 shows more detailed results, comparing times for the OpenMRS baseline and the *Phos*-instrumented version for each action (left graph) and page load (right graph). These results show that timing overhead is fairly consistent, albeit with some significant anomalies. In particular, overhead for the instrumented version spiked on the `dataExport.list` action, which is action number 23 in the graph. It is unclear what caused this anomaly, but appears to be an artifact of the Phosphor implementation (not our modifications).

Memory. Figure 18 shows baseline memory consumption during test runs of unmodified OpenMRS, versus pre-initialized OpenMRS instrumented with *Phos*. As these results demonstrate, while the instrumentation with *Phos* does impose memory overhead, the impact on performance is not practically significant.

7. Related Work and Conclusion

Some of the results in this paper were discussed in a preliminary manuscript [11], but the current work provides a fully developed metatheory, a formulation of the high-level security policy enforced by our system (explicit integrity modulo endorsement), and a complete implementation and empirical evaluation.

Taint analysis is an established solution to enforce confidentiality and integrity policies through direct data flow control. Various systems have been proposed for both low and high level level languages. Our policy language and semantics are based on a well-developed formal foundation, where we interpret Horn clause logic as an instance of information algebra [28] in order to specify and interpret retrospective policies. The work presented in this paper supersedes a previous presentation [11]– in the current paper we extend our language model, provide more rigorous proofs of correctness of policy enforcement, consider the hyperproperty of taint analysis in a model of Java, and report on a prototype implementation.

Schwartz et al. [1] define a general model for runtime enforcement of policies using taint tracking for an intermediate language. In Livshits et al. [8], taint analysis is expressed as part of operational semantics, similar to Schwartz et al., and a taxonomy of taint tracking is defined. Livshits et al. [2] propose a solution for a range of vulnerabilities regarding Java-based web applications, including SQL injections, XSS attacks and parameter tampering, and formalize taint propagation including sanitization. The work uses PQL [29] to specify vulnerabilities. However, these works are focused on operational definitions of taint analysis for imperative languages. In contrast we have developed a logical specification of taint analysis for a functional OO language model that is separate from code, and is used to establish correctness of an implementation. Our work also comprises a unique retrospective component to protect against incomplete input sanitization. According to earlier studies [2, 3], incomplete input sanitization makes a variety of applications susceptible to injection attacks.

Another related line of work is focused on the optimization of integrity taint tracking deployment in web-based applications. Sekar [30] proposes a taint tracking mechanism to mitigate injection attacks in web applications. The work focuses on input/output behavior of the application, and proposes a lower-overhead, language-independent and non-intrusive technique that can be deployed to track taint information for web applications by blackbox taint analysis with syntax-aware policies. In our work, however, we propose a deep instrumentation technique to enforce taint propagation in a layered in-depth fashion. Wei et al. [31] attempt to lower the memory overhead of TaintDroid taint tracker [32] for Android applications. The granularity of taint tracking places a significant role in the memory overhead. To this end, TaintDroid trades taint precision for better overhead, e.g., by having a single taint label for an array of elements. Our work reflects a more straightforward object-level taint approach in line with existing Java approaches.

Saxena et al. [33] employ static techniques to optimize dynamic taint tracking done by binary instrumentation, through the analysis of registers and stack frames. They observe that it is common for multiple local memory locations and registers to have the same taint value. A single taint tag is used for all such locations. A shadow stack is employed to retain the taint of objects in the stack. Cheng et al. [34] also study the solutions for taint tracking overhead for binary instrumentation. They propose a byte to byte mapping between the main and shadow memory that keeps taint information. Bosman et al. [35] propose a new emulator architecture for the x86 architecture from scratch with the sole purpose of minimizing the instructions needed to propagate taint. Similar to Cheng et al. [34], they use shadow memory to keep taint information, with a fixed offset from user memory space. Zhu et al. [36] track taint for confidentiality and privacy purposes. In case a sensitive input is leaked, the event is either logged, prohibited or replaced by some random value. We have modeled a similar technique for an OO language, through high level logical specification of shadow objects, so that each step of computation is simulated for the corresponding shadow expressions.

Particularly for Java, Chin et al. [22] propose taint tracking of Java web applications in order to prohibit injection attacks. To this end, they focus on strings as user inputs, and analyze the taint in character level. For each string, a separate taint tag is associated with each character of the string, indicating whether that character was derived from untrusted input. The instrumentation is only done on the string-related library classes to record taint information, and methods are modified in order to propagate taint information. Haldar et al. [23] propose an object-level tainting mechanism for Java strings. They study the same classes as the ones in Chin et al. [22], and instrument all methods in these classes that have some string parameters and return a string. Then, the returned value of an instrumented method is tainted if at least one of the argument strings is tainted. However, in contrast to our work, *only* strings are endowed with integrity information, whereas all values are assigned integrity labels in our approach. Recently Bodei et al. [37] have proposed a static enforcement mechanism for taint analysis in IoT devices which predicts the propagation of taint in the system according to the flow of control. These previous works lack retrospective features.

Recent work has also considered static analysis for ensuring proper *context-based* sanitization of user input data to defend against XSS attacks, in the JSPChecker system [38]. While this work refines what is meant by “correct” sanitization, it relies on static analysis and thus introduces false positives. In contrast, we propose a runtime tool that marks data generated by imperfect sanitizers for postfacto analysis. Our work is more general in the sense that it can be used for any category of integrity data flow vulnerabilities including XSS.

Phosphor [5, 6] is an attempt to apply taint tracking more generally in Java, to any primitive type and object class. Phosphor instruments the application and libraries at bytecode level based on a given list of taint source and sink methods. Input sanitizers with endorsement are not directly supported, however. As Phosphor avoids any modifications to the JVM, the instrumented code is still portable. Our work is an attempt to formalize Phosphor in FJ extended with input sanitization and in-depth enforcement. Our larger goal is to develop an implementation of in-depth dynamic integrity analysis for Java by leveraging the existing Phosphor system.

Secure information flow [39] and its interpretation as the well-known hyperproperty [9] of noninterference [40] is challenging to implement in practical settings [41] 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., [1, 8, 31, 42]), 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. [25] as a general model for information flow of confidential data, concentrated on cryptographic computations and key release (declassification [43]) and later employed in other data secrecy analyses [24, 44, 45]. Schoepe et al. [10] 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 propose a knowledge-based semantics, influenced by Volpano's weak secrecy [26] and gradual release [25]. 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, rather than restricting the discussion to direct information flow in a low level language, we model a high level OO language with a functional flavor to represent generality of our framework.

Schoepe et al. [46] have recently employed explicit secrecy to study correctness results for dynamic confidentiality taint analysis in a core imperative setting with pointers and I/O, and deployed a Java-based tool, called DroidFace. A recent framework by Balliu et al. [47] attempts to bring together the general information flow and direct flow analyses using a security condition that models indirect flows which are observable by a low confidentiality user.

A counterpart for attacker knowledge in the realm of general flow of information integrity, called attacker power [24], is introduced as the set of low integrity inputs that generate the same observables. In this regard, Askarov et al. [24] use holes in the syntax of program code for injection points, influenced by [48]. 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. [49] 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.

In addition to formal properties of direct information flow, our formulation of correctness conditions also considers a formalization of audit logging based on our previous work [7], which considered a safety property unrelated to taint analysis. Other authors have recently considered formal characterizations of auditing based on logics of justification [50, 51]. In contrast, we consider a specific security application of auditing in combination with taint analysis where audit logs are "extralinguistic" vestiges of program computation, whereas these related works consider programs that are able to reflect on their own audit trails, which is a distinct theoretical problem.

7.1. Conclusion

In this paper we considered integrity taint analysis in a pure object-oriented language model. Our security model accounts for sanitization methods that may be incomplete, a known problem in practice and one inspired by our study of the OpenMRS medical records software system. We proposed an in-depth security mechanism based on combining prospective measures (to support access control) and retrospective measures (to support auditing and accountability) that address incomplete sanitization. More precisely, we propose treating the results of sanitization as "partially" endorsed, or "maybe tainted", and allow maybe tainted values to be used in security sensitive operations but record such events in the audit log.

We developed a uniform security policy of dynamic integrity taint analysis that specifies both prospective and retrospective measures, separate from code. The specification is defined in terms of a logical interpretation of program traces and leverages techniques from information algebra, allowing prospective and retrospective measures to be characterized in a uniform and integrated manner. Since the specification is defined separate from code, we use it to establish provable correctness conditions for a rewriting algorithm that instruments in-depth integrity taint analysis. A rewriting approach supports development of tools that can be applied to legacy code without modifying language implementations.

Although our specification of dynamic integrity taint analysis with endorsement establishes correctness conditions for implementations, it is still operational in nature. We therefore developed the hyperproperty of explicit integrity modulo endorsement to characterize the security property of integrity taint analysis in a non-operational manner. It is important to note that this formulation was not simply the dualization of previous formulations of explicit secrecy [10], since these formulations address only low-level code with unstructured data. We subsequently demonstrated that the image of our rewriting algorithm enjoys this security property.

Since our broader goal is to support well-founded practical tools for hardening software, we developed an instrumented version of OpenMRS that integrates our in-depth taint analysis formally specified in our model. Results from our evaluation of this implementation suggest that it is correct and practically feasible. We have made the implementation available on a public GitHub [27].

Acknowledgments.. This work was supported by the National Science Foundation under Grant No. 1408801. Thanks to Ramy Koudsi and Adam Barson for their work on the implementation of *Phos*, and to Scott Smith for comments on early drafts of this work.

References

[1] 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.

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

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

[4] Vinod Ganapathy, Trent Jaeger, Christian Skalka, and Gang Tan. Assurance for defense in depth via retrofitting. In *LAW*, 2014.

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

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

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

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

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

[10] 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.

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

[12] J. Kohlas. *Information Algebras: Generic Structures For Inference*. Discrete mathematics and theoretical computer science. Springer, 2003.

[13] Christian Skalka, Sepehr Amir-Mohammadian, and Samuel Clark. Dynamic integrity taint analysis in depth. Technical report, University of Vermont, 2019. http://www.cs.uvm.edu/~ceskalka/skalka-pubs/phos-TR19.pdf.

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

[15] Juerg Kohlas and Juerg Schmid. An algebraic theory of information: An introduction and survey. *Information*, 5(2):219–254, 2014.

[16] Usage statistics module. https://wiki.openmrs.org/display/docs/Usage+Statistics+Module, 2010. Accessed: 2015-09-27.

[17] Sepehr Amir-Mohammadian. *A Formal Approach to Combining Prospective and Retrospective Security*. PhD thesis, The University of Vermont, 2017.

[18] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.

[19] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *CCS 2011*, pages 151–162, 2011.

[20] J. G. Cederquist, Ricardo Corin, M. A. C. Dekker, Sandro Etalle, J. I. den Hartog, and Gabriele Lenzini. Audit-based compliance control. *International Journal of Information Security*, 6(2-3):133–151, 2007.

[21] 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.

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

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

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

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

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

[27] Christian Skalka, Sepehr Amir-Mohammadian, and Samuel Clark. Retrospective Taint Analysis for OpenMRS. https://github.com/uvm-plaid/phosphor-mod, 2019.

[28] Juerg Kohlas and Juerg Schmid. An algebraic theory of information: An introduction and survey. *Information*, 5(2):219–254, 2014.

[29] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: A program query language. In *OOPSLA*, 2005.

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

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

[32] 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.

[33] Prateek Saxena, R. Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *CGO*, pages 74–83, 2008.

[34] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *IEEE ISCC*, pages 749–754, 2006.

[35] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *RAID*, pages 1–20, 2011.

[36] 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.

[37] Chiara Bodei and Letterio Galletta. Tracking sensitive and untrustworthy data in IoT. In *ITASEC*, pages 38–52, 2017.

[38] Antonin Steinhauser and François Gauthier. Jspchecker: Static detection of context-sensitive cross-site scripting flaws in legacy web applications. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 57–68, New York, NY, USA, 2016. ACM.

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

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

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

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

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

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

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

[46] Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Let's face it: Faceted values for taint tracking. In *European Symposium on Research in Computer Security*, pages 561–580, 2016.

[47] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. We are family: Relating information-flow trackers. In *European Symposium on Research in Computer Security*, pages 124–145, 2017.
[48] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.
[49] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *ICISS*, pages 48–65, 2010.
[50] Wilmer Ricciotti and James Cheney. Strongly normalizing audited computation. *CoRR*, abs/1706.03711, 2017.
[51] Francisco Bavera and Eduardo Bonelli. Justification logic and audited computation. *Journal of Logic and Computation*, page exv037, 2015.