In-Depth Enforcement of Dynamic Integrity Taint Analysis

Sepehr Amir-Mohammadian  
University of Vermont  
samirmoh@uvm.edu

Christian Skalka  
University of Vermont  
ceskalka@uvm.edu

ABSTRACT

Dynamic taint analysis can be 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. The rewriting itself is a model of existing, efficient Java taint analysis tools.

1. INTRODUCTION

Dynamic taint analysis implements a “direct” or “explicit” information flow analysis to support a variety of security mechanisms. Like information flow generally, taint analysis can be used to support either confidentiality or integrity properties. An important application of integrity taint analysis is to prevent the input of untrusted data to security sensitive operations, in particular to combat cross-site scripting (XSS) and SQL injection attacks in web applications [15]. Any untrusted user input is marked as tainted, and then taint is tracked and propagated through program values 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 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 [25, 15]. For instance in a real-world news management system [25], user input is supposed be a numerical value, but due to erroneous implementation of input sanitizer, the flawed sanitization mechanism admits a broader range of data. This results in SQL command injection vulnerability. We call such incomplete sanitizers partially trusted or imperfect throughout the paper.

Thus, a main challenge we address in this paper is how to mitigate imperfect sanitization in taint analysis. Our solution is an in-depth approach [10]—we propose to use a combination of prospective and retrospective measures to reduce false positives while still providing security measures in the presence of imperfect sanitization. We are concerned about both efficiency and correctness—our taint analysis model is intended to capture the essence of Phosphor [3, 4], an existing Java taint analysis system with empirically demonstrated efficiency, and our approach to retrospective security is intended to minimize the size of logs [1]. Theoretical foundations we establish in this paper are intended to support a Java rewriting algorithm that is specifically intended to instrument security in the legacy OpenMRS medical records system with acceptable performance overhead. This would extend upon our previous in-depth security tools for OpenMRS [1]. However, the formulations proposed here could be applied more broadly.

Uniform Policy Specification and Instrumentation Correctness.

An important feature of our approach is a uniform expression of an in-depth security policy, that combines the typical blocking (prospective) behavior of taint-based access control with audit logging (retrospective) features. Furthermore, our policy specification for both prospective and retrospective analyses are separate from code, rather than being embedded in it. Our policy language and semantics are based on a well-developed formal foundation explored in previous work [1]. This work also establishes correctness conditions for policy implementations. We propose a rewriting algorithm that instruments code with support for in-depth integrity taint analysis in the presence of partially trusted sanitization, and prove that it is correct with respect to the in-depth policy specification. This algorithm and its proof of correctness result is formulated with respect to an idealized language model. Implementation in a real-world general purpose language setting is a topic for future work.

1.1 Vulnerability and Countermeasures

Our work is significantly inspired by a previously unreported security flaw in OpenMRS [18], a Java-based open-source web application for medical records. This flaw allows an attacker to launch persistent XSS attacks1. When a web-based software receives and

---

1We have responsibly disclosed the vulnerabilities we have found.
stores user input without proper sanitization, and later, retrieves these information to (other) users, persistent XSS attacks could take place.

OpenMRS uses a set of validators to enforce expected data formats by implementation of the Validator interface (for instance, 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. In our example, 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 [25, 15].

To support integrity taint analysis in the presence of incomplete sanitization for legacy code, we propose a program rewriting approach, which is applicable to systems such as OpenMRS. 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 tainted sources, secure sinks, and sanitizers. A distinct feature of our system is that results of sanitization are considered “maybe tainted” data, which is allowed to flow into security sensitive operations but in such cases is entered in a log to support auditing and accountability.

Our rewriting algorithm is intended to be a pure model of the Phosphor taint analysis system [3, 4] to track direct information flow, augmented with endorsement and retrospective security measures. We are inspired by this system due to its proven efficiency and general security model allowing any primitive value to be assigned an integrity level (rather than just e.g. strings as in other systems [11, 8]). Foundational correctness of the algorithm is proven in an idealized model of Phosphor-style taint analysis defined for a “Featherweight” Java core language [12]. We choose this model to focus on basic issues related to “pure” object orientation control and data flow, e.g. method dispatch and propagation through value (de)construction. Correctness is established in terms of the faithfulness of the instrumentation to the policy specification.

### 1.2 The Security Model

The security problem we consider is about the integrity of data being passed to security sensitive operations (ssos). An important example is a string entered by an untrusted users that is passed to a database method for parsing and execution as a SQL command. The security mechanism should guarantee that low-integrity data cannot be passed to ssos without previous sanitization.

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. While there are no primitive conditional expressions in our Java model, indirect flows are realized via dynamic method dispatch which faithfully models Java dispatch.

More precisely, we posit that programs \( p(\theta) \) in this security setting contain a low integrity data source \( \theta \), and an arbitrary number of secure sinks (ssos) and sanitizers which are specified externally to the program by a security administrator. For simplicity we assume that ssos are unary operations, 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 in a class. That is, there exists a set of sanitizers containing class, method pairs \( \text{C.m} \) which are assumed to return high-integrity data, though they may be passed low-integrity data. Likewise, there exists a set of ssos of the same form, and for brevity we will write \( \text{ssos} \) for a method invocation \( \text{new o.m(s)} \) on some object \( o \) where \( C.m \in \text{SSO} \). As a sanity condition we require \( \text{SSOs} \cap \text{Sanitizers} = \emptyset \). For simplicity of our formal presentation we assume that only one tainted source will exist.

### The Threat Model.

We assume that our program rewriting algorithm is trusted. Input code is trusted to not be 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 won’t be exploited (even accidentally) as a side-channel attack vector by non-malicous 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 Outline of the Paper

In Section 2, we define the basic source language model. In Section 3, we introduce a logical specification of dynamic taint analysis with partial endorsement, that can be used to support a uniform specification of an in-depth (prospective and retrospective) policy. The main feature of this Section is Definition 3.6 which specifies the in-depth policy. In Section 4, an implementation of dynamic integrity taint analysis is defined as a program rewriting algorithm. This Section also contains an extended example in Section 4.4 illustrating the main ideas of our formulations. In Section 5 the rewriting algorithm is proven correct on the basis of the in-depth policy specification. The main results are that rewritten programs are simulations of the source with integrity flow reflected in the operational semantics (Theorem 5.1), and that prospective and retrospective policies are correctly instrumented (Theorems 5.2 and 5.3). We conclude with a discussion of related work in Section 6 and ideas for future work in Section 7.
2. AN OO MODEL FOR INTEGRITY TAINT ANALYSIS

We begin the technical presentation with definition of our language model based on Featherweight Java (FJ) [12]. FJ is a core calculus that includes class hierarchy definitions, subtyping, dynamic dispatch, and other basic features of Java. To this we add semantics for library methods that allow specification of operations on base values (such as strings and integers). Consideration of these features is important for a thorough modeling of Phosphor-style taint analysis, and important related issues such as string-vs. character-based taint [8] which have not been considered in previous formal work on taint analysis [22]. Since static analysis is not a topic of this paper, for brevity we omit the standard FJ type analysis which is described in [12].

2.1 Syntax

The syntax of FJ is defined below. 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(v1, ..., vn). 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.

L ::= class C extends C { C f; K R}
K ::= C(C f){super(f); this.f = f;}
M ::= C.m(C f){return e;}
e ::= x | e.f | e.m(u) | new C(v) | C.m(e)
E ::= [] | E.f | E.m(u) | v.m(v, E, e') | new C(v, E, e') | C.m(E)

For brevity in this syntax, we use vector notations. Specifically we write f to denote the sequence f1, ..., fn, similarly for C, m, x, etc., and we write R as shorthand for M1 · · · Mn. We write the empty sequence as ∅, we use a comma as a sequence concatenation operator. If and only if m is one of the names in m, we write m ∈ m. Vector notation is also used to abbreviate sequences of declarations; we let C f and C f; denote C1 f1, ..., Cn fn, and C1 f1; ...; Cn fn; respectively. The notation this.f = f; abbreviates this.f1 = f1; ...; this.fn = fn; Sequences of names and declarations are assumed to contain no duplicate names.

2.2 Semantics

The semantic definition has several components, in addition to evaluation rules.

2.2.1 The class table and field and method body lookup

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

\[
\text{fields}_{CT}(\text{Object}) = \emptyset
\]

\[
CT(C) = \text{class } C \text{ extends } D \{ C f; K R \}
\]

\[
\text{fields}_{CT}(D) = \emptyset
\]

\[
\text{fields}_{CT}(C) = D g, C f
\]

\[
CT(C) = \text{class } C \text{ extends } D \{ C f; K R \}
\]

\[
B \text m(B x)(\text{return } e; \} \in R
\]

\[
\text{mbody}_{CT}(m, C) = x, e
\]

\[
CT(C) = \text{class } C \text{ extends } D \{ C f; K R \}
\]

\[
m \notin R
\]

\[
\text{mbody}_{CT}(m, C) = \text{mbody}_{CT}(m, D)
\]

2.2.2 Method type lookup

Just as we’ve defined a function for looking up method bodies in the class table, we also define a function that will look up method types in a class table. Although we omit FJ type analysis from this presentation, method type lookup will be useful for taint analysis instrumentation (Definition 4.1).

\[
\text{class } C \text{ extends } D \{ C f; K R \}
\]

\[
\text{mbtype}_{CT}(m, C) = B \rightarrow B
\]

\[
\text{class } C \text{ extends } D \{ C f; K R \}
\]

\[
m \notin R
\]

\[
\text{mbtype}_{CT}(m, C) = \text{mbtype}_{CT}(m, D)
\]

2.2.3 Operational semantics

Now, we can define the operational semantics of FJ. We define these as a small step relation in the usual manner.

\[
\begin{align*}
\text{Invoke} & \quad \text{mbody}_{CT}(m, C) = x, e \\
\text{Return} & \quad \text{C.m(v) } \rightarrow v
\end{align*}
\]

We use →* to denote the reflexive, transitive closure of →. We will also use the notion of an execution trace τ to represent a series of configurations κ, where τ = κ1 · · · κn means that κi → κi+1 for 0 < i < n. In the case of FJ, configurations κ are just expressions e. 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.

To denote execution of top-level programs p(θ) where θ is an object of low integrity, we assume that all class tables CT include an entry point TopLevel.main, where TopLevel objects have no fields. We define p(θ) = new TopLevel().main(θ), and we write p(θ) ↓ τ iff trace τ begins with the configuration p(θ).

2.3 Library Methods

The abstract calculus described above is not particularly interesting with respect to direct information flow and integrity propagation, especially since method dispatch is considered an indirect flow. More interesting is the manner in which taint propagates through primitive values and library operations on them, especially strings and string operations. This is because direct flows should propagate through some of these methods. Also, for run-time efficiency and ease of coding some Java taint analysis tools treat even
complex library methods as “black boxes” that are instrumented at
the top level for efficiency [11], rather than relying on instrumenta-
tion of lower-level operations.

Note that treating library methods as “black boxes” introduces a potential for over- and under-tainting—e.g. 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 [11]. Clearly this strategy introduces a potential for over-taint. Other systems do not propagate taint from strings to their component characters when decomposed [8], which is an example of under-taint. Part of our goal here is to develop an adequate language model to consider these approaches.

We therefore extend our basic definitions to accommodate primitive values and their manipulation. Let a primitive field be a field containing a primitive value. We call a primitive object/class any object/class with primitive fields and, or a library method is any method that operates on primitive objects, defined in a primitive class. We expect primitive objects to be object wrappers for primitive values (e.g., 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 LibMethods be the set of library method names paired with their corresponding primitive class names.

We posit a special set of field names PrimField that access primitive values ranged over by \( \nu \) 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 \( \nu \) may occur in special object fields and be manipulated there by any \( Op \).

\[
\begin{align*}
\mathcal{F}^+ & \subseteq \text{PrimField} \\
\mathcal{E} & ::= \nu \mid \mathcal{E} \mathcal{F} \\
\mathcal{E} & ::= \cdots \mid \text{Op}(\mathcal{E}) \\
\mathcal{V} & ::= \text{new } \mathcal{C}(\mathcal{E}) \mid \nu \\
\mathcal{E} & ::= \cdots \mid \text{Op}(\mathcal{V}, \mathcal{E}, \mathcal{E})
\end{align*}
\]

For library methods we require that the body of any library method be of the form where \( \mathcal{C} \) is a primitive class:

\[
\text{return new } \mathcal{C}(\bar{s}_1, \ldots, \bar{s}_n)
\]

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 \( \text{Int} \) objects contain a primitive numeric \( \text{val} \) 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 \( \text{Int} \) in \( CT \) a method \( \text{Plus} \) to support arithmetic in programs:

\[
\text{Int plus(Int x)} \{ \text{return(new(Int)})(+(\text{this.val}, x.\text{val}))\};
\]

Similarly, to define string concatenation, we define a concatenation operation \( \circ \) on primitive strings:

\[
\circ(s_1, s_2) \approx s_1 s_2
\]

and we extend the definition of \( \text{String} \) in \( CT \) with the following method, where we assume all \( \text{String} \) objects maintain their primitive representation in a \( \text{val} \) field:

\[
\text{String concat(String x)} \{ \text{return(new(String)})(@this.\text{val}, x.\text{val})\};
\]

\section{In-Depth Integrity Analysis Specified Logically}

In this Section we demonstrate how in-depth integrity direct 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” expression that reflects the proper taint for all data in a configuration at any point in a trace.

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

\begin{example}
Suppose a method \( m \) of an untainted \( \mathcal{C} \) object with no fields is invoked on a pair of tainted \( s_1 \) and untainted \( s_2 \) strings: \( \text{new C().m(new String(s_1), new String(s_2))} \). The proper shadow is:

\[
\text{shadow C(\circ).m(\text{shadow String(\bullet), shadow String(\circ)})}.
\]

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 a later Section (4.4).
\end{example}

\subsection{Traces and Logs as Information}

Leveraging theory developed in previous work [1], we interpret traces and audit logs as elements in an information algebra [13] to support uniform policy specification and semantics for dynamic integrity taint analysis.

An information algebra contains information elements \( X \) (e.g. a set of logical assertions) taken from a set \( \Phi \) (the algebra). A partial ordering is induced on \( \Phi \) by the so-called information ordering relation \( \leq \), where intuitively for \( X, Y \in \Phi \) we have \( X \leq Y \) iff \( Y \) contains at least as much information as \( X \), though its precise meaning depends on the particular algebra. We say that \( X \) and \( Y \) are information equivalent, and write \( X \equiv Y \), iff \( X \leq Y \) and \( Y \leq X \). We assume given a function \( \lfloor - \rfloor \) that is an injective mapping from traces to \( \Phi \). This mapping interprets a given trace as information, where the injective requirement ensures that information is not lost in the interpretation. For example, if \( \sigma \) is a proper prefix of \( \tau \) and thus contains strictly less information, then formally \( \lfloor \sigma \rfloor \leq \lfloor \tau \rfloor \).

It is well known that safety properties such as prospective taint analysis can be described as sets of traces [20]. An information algebra formulation also supports a specification and semantics of retrospective policies [1]. Specifically, we let \( \text{LS} \) range over logging specifications, which are functions from traces to \( \Phi \). Intuitively, \( \text{LS}(\tau) \) denotes the information that should be recorded in an audit log during the execution of \( \tau \) given specification \( \Phi \). This is the semantics of the logging specification \( \text{LS} \). Assuming also an interpretation of logs \( \lfloor - \rfloor \) as a function from logs \( \mathbb{L} \) to information elements, this formulation establishes correctness conditions for audit logs as follows.
DEFINITION 3.1. Audit log \( L \) is sound with respect to logging specification \( LS \) and execution trace \( \tau \) iff \( |L| \leq LS(\tau) \). Similarly, audit log \( L \) is complete with respect to logging specification \( LS \) and execution trace \( \tau \) iff \( LS(\tau) \leq |L| \).

### 3.1.1 FOL as an Information Algebra

To define direct information integrity flow as a property of FJ traces, we use safe Horn clause logic (aka Datalog) extended with constructors as terms [6, 17], in keeping with our previous work on specification of retrospective policies [1]. In that work, we have demonstrated that this variant of first order logic (FOL) is an instance of information algebra [1].

Formally, we let Greek letters \( \phi \) and \( \psi \) range over FOL formulas over terms \( T \) that include variables, constants and compound terms, 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. The closure of a set of formulas \( X \) is the set of formulas that are logically entailed by \( X \).

DEFINITION 3.2. We define a closure operation \( C \), and a set \( \Phi_{FOL} \) of closed sets of formulas:

\[
C(X) = \{ \phi \mid X \vdash \phi \} \quad \Phi_{FOL} = \{ X \mid C(X) = X \}
\]

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

Let \( Preds \) be the set of all predicate symbols, and let \( S \subseteq Preds \) be a set of predicate symbols. We define sublanguage \( LS \) 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 \( S \), on the set of all sublanguages, with \( F = L_{Preds} \) as the top element.

Now we can define 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 3.3. Define:

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

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

### 3.2 Taint Tracking as a Logical Trace Property

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

**Figure 1:** Definition of \( toFOL(\cdot) \) for configurations.

- \( match(sv, \cdot) \)
- \( match(shadow C(t, \bar{w}), f, \cdot) \)
- \( match(shadow C(t, \bar{w}), \bar{m}(\bar{w}), \cdot) \)
- \( match(\bar{m}(sv), \cdot, \cdot) \)
- \( match(\bar{w}, SE, se') \Rightarrow match(se, t, SE, f, se') \)
- \( match(se, SE, se') \Rightarrow match(\bar{w}, m(\bar{w}), SE, m(\bar{w}), se') \)
- \( match(se, SE, se') \Rightarrow match(sv, m(\bar{w}, SE, \bar{w}, se), se') \)
- \( match(\bar{w}, se, se') \Rightarrow match(\bar{m}(sv), SE, se') \)
- \( match(\bar{w}, se, se') \Rightarrow match(Op(\bar{w}, se, se), Op(\bar{w}, SE, se), se') \)

**Figure 2:** match predicate definition.

DEFINITION 3.4. We define \( toFOL(\cdot) \) as a mapping on traces and configurations:

\[
\nu \in toFOL(\tau) = \bigcup_{\sigma \in predX(\tau)} \nu \cap toFOL(\sigma)
\]

such that \( toFOL(\sigma) = \bigcup_{\sigma \in predX(\tau)} \nu \cap toFOL(\kappa_i) \) for \( \sigma = \kappa_1 \cdots \kappa_k \). We define \( toFOL(\kappa, n) \) as in Figure 1.

#### 3.2.1 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 (\( \cdot \)), low integrity (\( \circ \)), and questionable integrity (\( \odot \)).

\[
t : = \circ | \odot | \bullet
\]

We specify an ordering \( \leq \) on these labels denoting their integrity relation:

\[
\bullet \leq \odot \leq \circ
\]

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 \( \land \) and join \( \lor \) operations on taint lattice elements, and introduce logical predicates meet and join such that meet \( (t_1 \land t_2, t_1, t_2) \) and join \( (t_1 \lor t_2, t_1, t_2) \) 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 manipulate shadow terms and context, which are terms \( T \) in the logic with the following syntax. Note the structural conformance with closed \( e \) and \( E \), but with primitive values replaced with a single dummy value \( \delta \) that is omitted for brevity in examples, but is necessary to maintain proper arity for field selection. Shadow expressions most importantly assign integrity identifiers \( t \) to objects:

\[
\begin{align*}
sv & ::= \text{shadow } C(t, \overline{s}v) \mid \delta \\
se & ::= se \mid se.\overline{e} \mid se.m(\overline{s}e) \mid \text{shadow } C(t, \overline{s}e) \mid C.m(se) \mid Op(\overline{s}e) \\
SE & ::= \{ \mid SE.\overline{e} \mid SE.m(\overline{s}e) \mid sv.m(\overline{s}v, SE, \overline{s}e') \mid \text{shadow } C(t, \overline{s}v, SE, \overline{s}e') \mid C.m(SE) \mid Op(\overline{s}v, SE, \overline{s}e) \}
\end{align*}
\]

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, \( \text{smbody} \), that imposes untainted tags on all method bodies, defined a priori, and removes primitive values.

**Definition 3.5.** Shadow method bodies are defined by the function \( \text{smbody} \).

\[
\text{smbody}_{C,T}(m, C) = \#.\text{smrewrite}(e),
\]

where \( mbody_{C,T}(m, C) = \#.e \) and the shadow rewriting function, \( \text{smrewrite} \), is defined as follows, where \( \text{smrewrite}(\#) \) denotes a mapping of \( \text{smrewrite} \) over the vector \( \# \):

\[
\begin{align*}
\text{smrewrite}(x) & = x \\
\text{smrewrite}(\text{new } C(\#)) & = \text{shadow } C(o, \text{smrewrite}(\#)) \\
\text{smrewrite}(e.f) & = \text{smrewrite}(e).f \\
\text{smrewrite}(e.m(\#')) & = \text{smrewrite}(e).m(\text{smrewrite}(\#')) \\
\text{smrewrite}(C.m(\#)) & = C.m(\text{smrewrite}(\#)) \\
\text{smrewrite}(Op(\#)) & = Op(\text{smrewrite}(\#)) \\
\text{smrewrite}(\nu) & = \delta
\end{align*}
\]

We use match as a predicate which matches a shadow expression \( se \), to a shadow context \( SE \) and a shadow expression \( se' \) where \( se' \) is the part of the shadow in the hole. The definition of match is given in Figure 2.

Next, in Figure 3, we define a predicate \( \text{Shadow}(n, se) \) where \( se \) is the relevant shadow expression at execution step \( n \), establishing an ordering for the shadow trace. Shadow has as its precondition a “current” shadow expression, and as its postcondition the shadow expression for the next step of evaluation (with the exception of the rule for shadowing \( Op \)s on primitive values which reflects the “immediate” valuation due to the definition of \( se \)– note the timestamp is not incremented in the postcondition in that case).

We set the shadow of the initial configuration at timestamp 1, and then \( \text{Shadow} \) inductively shadows the full trace. \( \text{Shadow} \) is defined by case analysis on the structure of shadow expression in the hole. The shadow expression in the hole and the shadow evaluation context are derived from match predicate definition.\(^2\)

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

\[
\text{meet}(t, t_1, t_2) \Rightarrow \text{Prop}(t, \text{Int}.\text{plus}(t_1, t_2))
\]

### 3.3 In-Depth Integrity Taint Analysis Policies

We define our in-depth policy for integrity taint analysis. The prospective component blocks the execution of the program whenever a tainted value is passed to a secure method. To this end, in Figure 4 we define the predicate \( \text{BAD} \) which identifies traces that should be rejected as unsafe. 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 \( \text{MaybeBad} \) also defined in Figure 4.

**Definition 3.6.** Let \( X \) be the set of rules in Figures 2, 3, and 4 and the set of user defined rules for \( \text{Prop} \). The prospective integrity taint analysis policy is defined as the set of traces that do not end in \( \text{BAD} \) states.

\[
\text{SP}_{\text{taint}} = \{ \tau \mid (\tau \in C(X)) \Rightarrow (\text{BAD}) = C(\emptyset) \}.
\]

The retrospective integrity taint analysis policy is the following logging specification:

\[
\text{LS}_{\text{taint}} = \lambda \tau.((\tau \in C(X)) \Rightarrow (\text{MaybeBAD})
\]

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

**Definition 3.7.** We call a program \( p(\theta) \) safe iff for all \( t \) it is the case that \( p(\theta) \downarrow \tau \) implies \( \tau \in \text{SP}_{\text{taint}} \). We call the program unsafe iff there exists some trace \( \tau \) such that \( p(\theta) \downarrow \tau \) and \( \tau \notin \text{SP}_{\text{taint}} \).

### 4. TAINT ANALYSIS INSTRUMENTATION VIA PROGRAM REWRITING

Now we define an object based dynamic integrity taint analysis in a more familiar operational style. Taint analysis instrumentation is added automatically by a program rewriting algorithm \( R \) that models the Phosphor rewriting algorithm. 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.

#### 4.1 In-Depth Taint Analysis Instrumentation

The target language of the rewriting algorithm \( R \), called \( \text{FJ}_{\text{taunt}} \), is the same as \( \text{FJ} \), except we add taint labels \( t \) as a form of primitive value \( v \), the type of which we posit as \( \text{Taint} \). For the semantics of taint values operations we define:

\[
\begin{align*}
\forall (t_1, t_2) \approx t_1 \lor t_2 \quad & \land (t_1, t_2) \approx t_1 \land t_2
\end{align*}
\]

In addition we introduce a “check” operation \( ? \) such that \( ? t \approx t \) iff \( t \neq \bullet \). We also add an explicit sequencing operation of the form \( e.\overline{e} \rightarrow \text{target language expressions}, and evaluation contexts of the form \( E.\overline{e} \rightarrow \bullet \), along with the appropriate operational semantics rule that we define below in Section 4.3.

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

\[\footnote{Some notational liberties are taken in Figure 3 regarding expression and context substitutions, which are defined using predicates elided for brevity.}\]
blocks if the argument is tainted, an \textit{endorse} method for any object class returned by a sanitizer, and modification of all sanitizers to \textit{endorse} their return value.

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 3.6, we define \textit{endorse} so it takes object taint \textit{t} to the join of \(t \land \ominus\). The algorithm also adds a \textit{log} method call to the beginning of \textit{SSOs}, which will log objects that are maybe tainted or worse. The semantics of \textit{log} are defined directly in the operational semantics of \textit{FJ\textsubscript{taint}} below.

\begin{definition}
For any expression \(e\), the expression \(\mu(e)\) is syntactically equivalent to \(e\) except with every subexpression \textit{new C(\(\bar{e}\))} replaced with \textit{new C(\(e, \bar{e}\))}. Given \textit{SSOs} and \textit{Sanitizers}, define \(R(e, CT) = (\mu(e), R(CT))\), where \(R(CT)\) is the smallest class table satisfying the axioms given in Figure 5.
\end{definition}

4.2 Taint Propagation of Library Methods

Another important element of taint analysis is instrumentation of library methods that propagate taint— the propagation must be made explicit to reflect the interference of arguments with results. The approach to this in taint analysis systems is often motivated by efficiency as much as correctness [11]. We assume that library methods are instrumented to propagate taint as intended (i.e. in accordance with the user defined predicate \textit{Prop}).

Here is how addition and string concatenation, for example, 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:

\begin{verbatim}
Int plus(Int x)
  { return(new(Int)
     \land(this.taint, x.taint), +(this.val, x.val)); }

String concat(Int x)
  { return(new(String)
     \land(this.taint, x.taint), @[this.val, x.val]); }
\end{verbatim}

4.3 Operational Semantics of \textit{FJ\textsubscript{taint}}

To support the semantics of \textit{log}, we add an audit \textit{log} \(L\) as a new configuration component in \textit{FJ\textsubscript{taint}} that stores objects of questionable integrity. The \textit{log} method is the only one that interacts with the \textit{log} in any way. We “inherit” the reduction semantics of \textit{FJ}, and add a rule also for evaluation of sequencing.

\begin{figure}
\centering
\begin{align*}
\text{match}(se, SE, \text{shadow } C(t, \bar{t}).m(\text{shadow } D(t', \bar{t}'))) \land \text{Shadow}(n, se) \land \text{Call}(n, C, v, m, u) \land m \in \text{SSOs} \implies \text{SsoTaint}(n, t', u).
\end{align*}
\caption{Predicates for Specifying Prospective and Retrospective Properties}
\end{figure}

\begin{center}
\begin{tabular}{c c}
Reduce & Sequence \[ e \rightarrow e' \]
\end{tabular}
\end{center}

As for \textit{FJ} we use \(\tau\) to denote the reflexive, transitive closure on \(\rightarrow\) over \textit{FJ\textsubscript{taint}} configurations of the form \(e, L\). We define \textit{FJ\textsubscript{taint}} configurations and traces as for \textit{FJ}. Abusing notation, we write \(R(p(\theta)) \downarrow \tau\) iff \(\tau\) begins with the configuration \(R(p(\theta))\), \(\ominus\), and also \(\kappa \downarrow \tau\) iff \(\tau\) is a valid trace in the \textit{FJ\textsubscript{taint}} semantics beginning with \(\kappa\).

4.4 An Illustrative Example

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

\begin{example}
Let
\begin{verbatim}
mbody\textsubscript{CT}(main, TopLevel) =
  x, new Sec(.).secureMeth(new Sec()).sanitize(x.concat(
    new String("world")));
\end{verbatim}

Assume the string "hello" is tainted with low integrity. Figure 6 gives the source trace, the shadow expressions derived based on the rules given in Figure 3, and the target trace. For the sake of brevity and clarity in illustrating the main ideas, we have assumed that
methods Sec.sanitize and Sec.secureMeth are identity functions. Some reduction steps are elided in the example as n-length multi-step reductions \( \rightarrow^* \). All reductions are provided in a complete version of the example in the Appendix (Example A.1).

5. CORRECTNESS OF PROGRAM REWRITING

The logical definition of in-depth integrity taint analysis presented in Section 3 establishes the proper specification of prospective and retrospective analysis. In this section we show how these definitions are used to establish correctness conditions for \( \mathcal{R} \), and how correctness is proven. The main properties defined in this Section establish correctness for prospective and retrospective measures in Definitions 5.3 and 5.5 respectively, and the main results demonstrate that these properties are enjoyed by \( \mathcal{R} \) in Theorems 5.2 and 5.3.

5.1 Semantics Preservation

A core condition for correctness of \( \mathcal{R} \) is proof of semantics preservation for safe programs in FJ, i.e. that rewritten programs simulate the semantics of source program modulo security instrumentations. The way this simulation is defined will naturally imply a full and faithful implementation of taint shadowing semantics. Adapting the Definition in [1], we say that rewriting algorithm \( \mathcal{R} \) is semantics preserving for SP\(_{\text{saint}} \) iff there exists a relation \( \approx \) with the following property.

**Definition 5.1.** Rewriting algorithm \( \mathcal{R} \) is semantics preserving iff for all safe programs \( p(\theta) \) (Definition 3.7) all of the following hold:

1. For all traces \( \tau \) such that \( p(\theta) \Downarrow \tau \) there exists \( \tau' \) with \( \tau \approx \tau' \) and \( \mathcal{R}(p(\theta)) \Downarrow \tau' \).
2. For all traces \( \tau \) such that \( \mathcal{R}(p(\theta)) \Downarrow \tau \) there exists a trace \( \tau' \) such that \( \tau' \approx \tau \) and \( p(\theta) \Downarrow \tau' \).

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

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

**Lemma 5.1.** \( \mathcal{R}(p(\theta)) \Rightarrow \mathcal{R}(p)(\mathcal{R}(\theta)) \)

5.2 Correctness for Prospective Analysis

Proof of semantics preservation establishes correctness for the prospective component of \( \mathcal{R} \), since SP\(_{\text{saint}} \) expresses the correct prospective specification as a safety property. To this end, we define the notion of security failure.

**Definition 5.2.** An FJ\(_{\text{saint}} \) program \( e \) causes a security failure iff \( e, \emptyset \rightarrow^* E[\bar{v}.\text{check}(\text{new } C(\bar{e}, \bar{v}))], \emptyset \) for some \( E, v, \text{ new } C(\bar{e}, \bar{v}) \), and \( L \).

The correctness of prospective component of rewriting algorithm is then defined as follows:

**Definition 5.3.** We call rewriting algorithm \( \mathcal{R} \) prospectively correct provided that a program \( p(\theta) \) is unsafe (Definition 3.7) iff \( \mathcal{R}(p(\theta)) \) causes a security failure (Definition 5.2).

5.3 Correctness for Retrospective Analysis

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 \( \tau \) describes execution of a source program, rewriting should produce a program with a trace \( \tau' \) that corresponds to \( \tau \) (i.e., \( \tau \approx \tau' \)), where the log \( L \) generated by \( \tau' \), written \( \tau' \prec L \), ideally contains the same information as \( LS(\tau) \). A minor technical issue is that instrumentation imposed by \( \mathcal{R} \) requires that information is added to the log after an sso invocation with an argument of at most questionable integrity, and \( \approx \) accounts for this stuttering. In our trace based correctness condition we need to account for this, hence the following Definition:

**Definition 5.4.** For FJ\(_{\text{saint}} \) programs we write \( \tau \prec L \) iff tail(\( \sigma \)) = \( e, L \) where \( \sigma \) is the longest trace such that \( \tau' \approx \tau \) and \( \tau' \prec L \) for some FJ trace \( \tau' \).

The following definitions then establish correctness conditions for rewriting algorithms. Note that satisfaction of either of these conditions only implies condition (1) of Definition 5.1, not condition (2), so semantics preservation is an independent condition. We define toFOL(\( L \)) = \{MaybeBAD(\( v \) ) | \( v \in L \)}, and thus \( \|L\| = C(\text{toFOL}(L)) \).

**Definition 5.5.** Rewriting algorithm \( \mathcal{R} \) is retrospectively sound/complete iff for all programs \( \mathcal{R}(p(\theta)) \), and finite traces \( \tau \) and \( \sigma \) where:

\[ \mathcal{R}(p(\theta)) \Downarrow \sigma \quad \tau \approx \sigma \quad \sigma \prec L \]

we have that \( L \) is sound/complete with respect to \( LS_{\text{saint}} \) and \( \tau \).
5.4 Definition of : ∼ and Correctness Results

To establish correctness of program rewriting, we need to define a correspondence relation : ∼. Source language execution traces and target language execution traces correspond if they represent the same expression evaluated to the same point. We make two special cases: when the source execution is about to return a value from a sanitization method that the target execution will endorse first, and the other case is where a sink method is called in the source execution, in which the target execution needs to first check the arguments to the sink method in order to log and enforce prospective policy by check. In these cases, the target execution may be ahead by some number of steps, allowing time to enforce heterogeneous policies.

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

**Definition 5.6.** The mapping overlay : (e, se) → e′ is defined in Figure 7.

We define a way to obtain the last shadow in a trace. Give a source trace τ of length n, LastShadow(τ) denotes the shadow of the last configuration in the trace τ. Considering the rule

\[ \text{Shadow}(n, se) \implies \text{LShadow}(se), \quad (1) \]

we define LastShadow(τ) = se such that [τ] ⊗ X ⊢ LShadow(se), where X contains the rules given in Figure 2, Figure 3 and (1).

We need to show that LastShadow is total function on non-trivial traces, i.e., LastShadow uniquely maps any non-empty trace to a shadow expression.

**Lemma 5.2.** LastShadow is total function on non-trivial traces.

**Proof.** By induction on the length of traces and the fact that shadow expressions are defined uniquely for every step of reduction in Figure 3.

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

**Definition 5.7.** The mapping trim : e → e′ is defined in Figure 8. We assume e to be no-op., i.e., e :: e = e.

**Definition 5.8.** Given source language execution trace τ = σκ and target language execution trace τ′ = σ′κ′, τ ≈ τ′ iff overlay(κ, LastShadow(τ)) = trim(e′), where κ′ = e′, L.
Theorem 5.1. The rewriting algorithm \( R \) is semantics preserving (Definition 5.1).

Theorem 5.2. The rewriting algorithm \( R \) is prospectively correct (Definition 5.3).

Theorem 5.3. The rewriting algorithm \( R \) is retrospectively sound and complete (Definition 5.5).

6. RELATED WORK

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 explored in previous work [1], where we interpret Horn clause logic as an instance of information algebra [13] in order to specify and interpret retrospective policies.

Schwartz et al. [22] define a general model for runtime enforcement of policies using taint tracking for an intermediate language. In Livshits [14], taint analysis is expressed as part of operational semantics, similar to Schwartz et al. [22], and a taxonomy of taint tracking is defined. Livshits et al. [15] 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 [16] 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 [25, 15], incomplete input sanitization makes a variety of applications susceptible to injection attacks.

In other work on taint analysis foundations, Schoepe et al. [21] have recently proposed a knowledge-based semantic notion for correctness of explicit information flow analysis, influenced by Volpano’s weak secrecy [24] and gradual release [2]. This work aims to develop a security hyperproperty, related to noninterference, that is enforced by taint analysis. However, this work is focused on confidentiality taint analysis in low level memory based languages. A similar property for integrity taint analysis in our language setting is a compelling topic for future work.

Another related line of work is focused on the optimization of integrity taint tracking deployment in web-based applications. Sekar [23] 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. [26] attempt to lower the memory overhead of TaintDroid taint tracker [9] 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 keeping with existing Java approaches.

Saxena et al. [19] 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. [7] 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. [5] 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. [7], they use shadow memory to keep taint information, with a fixed offset from user memory space. Zhu et al. [27] 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. [8] 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. Halder et al. [11] propose an object-level tainting mechanism for Java strings. They study the same classes as the ones in Chin et al. [8], and instrument all methods in these classes that have some string parameters and return a string. Then, the returned value of 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. These previous works also lack retrospective features.

Phosphor [3, 4] 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.

7. CONCLUSION AND FUTURE WORK

In this paper we have 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. We propose 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, in particular we consider sanitization results to be prospectively endorsed, but retrospectively tainted. We develop a uniform security policy that specifies both prospective and retrospective measures. This policy is used 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.

In future work, we aim to extend our existing rewriting tool for Java bytecode [1] to support in-depth integrity taint analysis in OpenMRS to harden against XSS and injection attacks. This tool
would incorporate the Phosphor system [3, 4] to achieve efficiency in the implementation (a common challenge for taint analyses), and would reflect the model studied in this paper. We are also interested in exploring hyperproperties of integrity taint analysis and its higher-level semantics in our language model, especially in relation to explicit secrecy as proposed for confidentiality taint analysis in recent work [21].

8. ACKNOWLEDGEMENTS
This work has been supported in part by the National Science Foundation under Grant No. 1408801.

9. REFERENCES


APPENDIX

A. AN ILLUSTRATIVE EXAMPLE
Here we demonstrate the example discussed in Section 4.4 with all reduction steps given, and the corresponding shadow expressions derived from the rules in Figure 3.

EXAMPLE A.1. Similar to Example 4.1 let $mbody_{CT}(main,TopLevel) =$ $x,new\text{Sec}().\text{secureMeth}(\text{newSec}).\text{sanitize}(x.concat("world"))$, and assume the string “hello” is tainted with low integrity. Figure 9 gives the FJ trace, the shadow expressions and the FJ$_{\text{taint}}$ trace.

B. PROOF OF CORRECTNESS

B.1 Proof of Semantics Preservation
In what follows, we prove the semantics preservation given by Definition 5.1. To this end, in Lemma B.1, we show that if trim of an expression is a value, that expression eventually reduces to that value provided it is not a security failure. Moreover, Lemma B.2 states that if trim of a non-security failure expression $e$ is reduced to $e'$ then $e$ reduces (in potentially multiple steps) to some expression with the same trim as $e'$.

LEMMA B.1. For all expressions $e$, if trim($e$) = $v$, then either (1) there exists some trace $\tau$ and log $\mathcal{L}$ such that $e$, $\mathcal{L} \vdash \tau \kappa$ where $\kappa = v$, $\mathcal{L}'$ for some log $\mathcal{L}'$, or (2) $e$ causes a security failure.

PROOF. By induction on the structure of $e$. □
Lemma B.2. For all expressions e, if trim(e), L \rightarrow e', L' then either (1) there exists σ such that e, L \downarrow σκ with κ = e'', L'' and trim(e'') = trim(e'), or (2) e causes a security failure.


Lemma B.3 states that overlaying a method body with its shadow is equal to the same method body in the rewritten class table.

Lemma B.3. overlay(e, securewrite(e)) = μ(e).

Proof. By induction on the structure of e. □

Lemma B.4 and Lemma B.5 state that single step and multi step reductions in FJ preserve ≈.

Lemma B.4. If τ1e1 :≈ τ2κ2 and e1 → e1' then there exists σ such that κ2 \downarrow σ and τ1e1e1' :≈ τ2σ.

Proof. By induction on the derivation of e1 → e1' and applying Lemmas B.2 and B.3. □

Lemma B.5. If τ1e1 :≈ τ2κ2 and e1 \downarrow σ1, then there exists σ2 such that κ2 \downarrow σ2 and τ1σ1 :≈ τ2σ2.

Proof. By induction on the derivation of e1 \downarrow σ1 and applying Lemma B.4. □

Similarly, Lemma B.6 and Lemma B.7 argue that single step and multi step reductions in FJtaint preserve ≈.

Lemma B.6. If τ1e1 :≈ τ2κ2 and κ2 → κ2' then there exists σ where e1 \downarrow σ and τ1σ :≈ τ1κ2κ2'.

Proof. By induction on the derivation of κ2 → κ2' and applying Lemma B.3. □

Lemma B.7. If τ1e1 :≈ τ2κ2 and κ2 \downarrow σ2, then there exists σ1 such that e1 \downarrow σ1 and τ1σ1 :≈ τ2σ2.

Proof. By induction on the derivation of κ2 \downarrow σ2 and applying Lemma B.6. □

Lemma B.8 states that initial configuration in FJtaint corresponds to the initial configuration in FJ. Finally, in Theorem 5.1, the semantics preservation property is proven.

Lemma B.8. Let \theta = new C(p) be a tainted input. Then, new TopLevel().main(\theta) :≈

new TopLevel(\theta).main(new C(\theta, p)), O.

Proof. By the definition of shadow expressions and :≈. □

Proof of Theorem 5.1.

Proof. Lemma B.8 states that initial configuration of a program p corresponds to the initial configuration of \mathcal{R}(p). Lemmas B.5 and B.7 extend the correspondence relation for traces of arbitrary lengths. More specifically, Lemmas B.8 and B.5 entail the 1st condition of Definition 5.1, and Lemmas B.8 and B.7 result in the 2nd condition of Definition 5.1. □

B.2 Proof of Prospective Correctness of Rewriting Algorithm

In order to prove that prospective component of \mathcal{R} is correct (Theorem 5.2), we first need to show that SPtaint is a safety property.

Lemma B.9. SPtaint is a safety property.

Proof. Let τ \notin SPtaint. Then, (SI τ \bigcap C(X)) \Rightarrow (BAD) \neq C(\emptyset). This implies that there exists some n such that BAD(n) \in (SI τ \bigcap C(X)) \Rightarrow (BAD). Let τ[1...n] denote the finite prefix of τ up to timestamp n. By Definition 3.6 BAD only refers to events that precede step n, so it follows that SI τ \bigcap C(X) \Rightarrow BAD(n) iff \tau[1...n] \bigcap C(X) \Rightarrow BAD(n), i.e. τ \notin SPtaint iff τ[1...n] \notin SPtaint for finite n, hence SPtaint is a safety property [20]. □

Proof of Theorem 5.2.

Proof. Suppose on the one hand that p(\theta) is unsafe, which is the case if p(\theta) \downarrow σ and τ \notin SPtaint for some τ. Then according to Lemma B.9, there exists some timestamp n such that τ[1...n] characterizes SPtaint, i.e., BAD(n) is derivable from the rules in Definition 3.6 and so (SI τ \bigcap C(X)) \Rightarrow (BAD). Let n be the least timestamp with such property and τ' = τ[1...n-1], and τ[n] = E[v.m(new D(\theta))] where v.m is an ssO invocation and new D(\theta) has low integrity by the Definition 3.6. By Theorem 5.1 there exists some trace σ such that \mathcal{R}(p(\theta)) \downarrow σ and τ' :≈ σ. Therefore, by definition of :≈ and \rightarrow we may assert that tail(σ) :≈ κ1, κ2, κ3 such that κ1 = E[v.m(new D(\theta))], κ2 = E[v.\log(new D(\theta))]; v.check(new D(\theta, \theta)), κ3 = E[v.check(new D(\theta, \theta))].

Thus, \mathcal{R}(p(\theta)) \downarrow σκ1κ2κ3 and therefore \mathcal{R}(p(\theta)) causes a security failure. Supposing on the other hand that \mathcal{R}(p(\theta)) causes a security failure, it follows that p(\theta) is unsafe by similar reasoning (i.e. fundamental appeal to Theorem 5.1), since security checks are only added to the beginning of ssO and fail if the argument has low integrity. □

B.3 Proof of Retrospective Correctness of Rewriting Algorithm

In what follows, we give a proof for retrospective soundness-completeness of rewriting algorithm \mathcal{R}, given in Definition 5.5. As in [1], our strategy is based on an appeal to least Herbrand models \mathcal{H} of the logging specifications and logs (least Herbrand models are known to exist for safe Horn clause logics with compound terms [17]). In essence, we demonstrate that audit logs generated by FJtaint programs are the least Herbrand model of the logging specification for the source program, hence contain the same information.

First, in Lemma B.10 and Lemma B.11, we assert propositions that hold for the syntactic property of closure in logics with a least Herbrand model [17].

Lemma B.10. C(\mathcal{H}(X)) = C(X) and \mathcal{H}(X) = \mathcal{H}(C(X)).

Lemma B.11. C(C(\mathcal{H}(X)) \cap L) = C(\mathcal{H}(X) \cap L).

We study the cases where a record is added to the audit log for single step (Lemma B.12) and multi step (Lemma B.13) reductions.
Lemma B.12. Let $e, L \rightarrow e', L'$. If $v \in L' - L$ then we have $e = E[u, \log(v)]$ for some evaluation context $E$ and value $u$, $v = new\ C(t, \overrightarrow{v})$ and $t \leq \circ$.

Proof. By induction on the derivation of $e, L \rightarrow e', L'$. \hfill \Box

Lemma B.13. Let $e, L \vdash \kappa, \kappa = e', L'$. If $v \in L' - L$ then there exists some trace $\sigma' \kappa_1 \kappa_2$ as the prefix of $\sigma$ such that $\kappa_1 = e_1, L_1, \kappa_2 = e_2, L_2$, where $e_1 = E[u, \log(v)]$ for some evaluation context $E$ and value $u$, $v = new\ C(t, \overrightarrow{v})$ and $t \leq \circ$.

Proof. By induction on the derivation of $e, L \vdash \kappa$ and applying Lemma B.12. \hfill \Box

Lemmata B.14, B.15 and B.16 extend the results of Lemmata B.2, B.4 and B.5 respectively, for $E_{taint}$ traces with maximal length.

Lemma B.14. For all expressions $e$, if $trim(e), L \rightarrow e', L'$ then either (1) there exists a $\kappa$ such that $e, L \vdash \sigma \kappa$ with $\kappa = e''$, $L''$, $trim(e'') = trim(e')$ and $\kappa \rightarrow \hat{\kappa}, L$ for some $\hat{\kappa}$ and $L$ implies $trim(\hat{\kappa}) \neq trim(e')$ or (2) $e$ causes a security failure.

Proof. By induction on the structure of $e$, and applying Lemma B.1. \hfill \Box

Lemma B.15. If $\tau_1 e_1 \approx \tau_2 \kappa_1 \kappa_2$ and $e_1 \rightarrow e'_1$, then there exists $\sigma$ such that $\kappa_2 \vdash \sigma, \tau_1 e_1 e'_1 \vdash \tau_2 \sigma$, and if $tail(\sigma) = \kappa_2$ and $\kappa_2 \rightarrow \kappa'_2$ for some $\kappa'_2$ then $\tau_1 e_1 e'_1 \not\vdash \tau_2 \sigma \kappa'_2$.

Proof. By induction on the derivation of $e_1 \rightarrow e'_1$ and applying Lemmata B.14 and B.3. \hfill \Box

Lemma B.16. If $\tau_1 e_1 \approx \tau_2 \kappa_1 \kappa_2$ and $e_1 \vdash \sigma_1$ non-trivially, then there exists $\kappa_2$ such that $\kappa_2 \vdash \sigma_2, \tau_1 \kappa_2 \vdash \tau_2 \kappa_2$, and if $tail(\sigma_2) = \kappa'_2$ and $\kappa'_2 \rightarrow \kappa''_2$ for some $\kappa''_2$ then $\tau_1 \kappa_2 \not\vdash \tau_2 \kappa''_2$.

Proof. By induction on the derivation of $e_1 \vdash \sigma_1$ and applying Lemma B.15. \hfill \Box

In Lemma B.17, we establish that the log generated by the rewritten program is the least Herbrand model of the given logging specification semantics. This allows us to easily prove retroactive correctness in Theorem 5.3.

Lemma B.17. Given $R(p(\theta)) \vdash \sigma$ and $\tau \vdash \sigma \sigma$ and $\tau \vdash L$, we have:

$$toFOL(L) = \delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}.$$  

Proof. (Sketch) We first show that $toFOL(L)$ is a subset of $\delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}$. Let $MaybeBAD(v) \in toFOL(L)$. According to the definition of $toFOL(L)$, $v \in L$. Using Lemma B.13, there exists some trace $\sigma' \kappa_1 \kappa_2$ as the prefix of $\sigma$ such that $\kappa_1 = e_1, L_1, \kappa_2 = e_2, L_2$, where $e_1 = E[u, \log(v)]$ for some evaluation context $E$ and value $u$, $v = new\ C(t, \overrightarrow{v})$ and $t \leq \circ$. Using Theorem 5.1, there exists a trace $\tau$ such that $p(\theta) \vdash \tau$ and $\tau \vdash \sigma' \kappa_1 \kappa_2$. This ensures that $MaybeBAD(v) \in \delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}$.

Next, we show that $\delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}$ is a subset of $toFOL(L)$. Let $MaybeBAD(new\ D(t', \overrightarrow{v})) \in \delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}$ and $v = new\ D(\overrightarrow{v})$. Then, there exist some $n, C, \overrightarrow{u}$ and $\overrightarrow{v}$ where Call$(n, C, \overrightarrow{u}, \overrightarrow{v}) \in toFOL(\tau)$. Moreover, there exist $SE, se, t, \overrightarrow{p}$ and $\overrightarrow{se'}$, where $Shadow(n, se)$ and match$(se, SE, shadow\ C(t, \overrightarrow{p}), m(shadow\ D(t', \overrightarrow{se'})))$ are derivable from the rules in $X, C \in SSOs$ and $t' \leq \circ$. Let $\tau'[\ldots \cdot n]$ denote the prefix of $\tau$ ending in timestamp $n$. Based on the definition of $toFOL(\tau)$, we can infer that $tail(\tau'[\ldots \cdot n]) = new\ C(\overrightarrow{u}, m(\overrightarrow{v}))$.

Using Theorem 5.1, we know that there exists trace $\sigma'$ such that $R(p(\theta)) \vdash \sigma'$ and $\tau'[\ldots \cdot n] \approx \sigma'$. Let $tail(\sigma') = \hat{\kappa}, L$. Therefore,

$$trim(\hat{\kappa}) = overlay(E[new\ C(\overrightarrow{u}, m(new\ D(\overrightarrow{v})))], \hat{SE}[[shadow\ C(t, \overrightarrow{p}), m(new\ D(t', \overrightarrow{se'}))]]) = E[new\ C(t, \overrightarrow{u}), m(new\ D(\overrightarrow{v})), L].$$

Obviously, $trim(\hat{\kappa}) \rightarrow E[new\ C(t, \overrightarrow{u}), log(new\ D(t', \overrightarrow{v})); e], L$ according to the semantics of target language. Then, using Lemma B.2, $tail(\sigma') \vdash \sigma' \kappa_1 \kappa_2 \kappa_3$, where

$$\kappa_1 = \hat{E}'[new\ C(t, \overrightarrow{u}), m(new\ D(\overrightarrow{v})), L'], L'$$
$$\kappa_2 = \hat{E}'[new\ C(t, \overrightarrow{u}), log(new\ D(t', \overrightarrow{v})); e], L', L'$$
$$\kappa_3 = \hat{E}'[e], L', L' \cap (new\ D(t', \overrightarrow{v})),$$

for some $\hat{E}'$ such that

$$trim(\hat{E}' new\ C(t, \overrightarrow{u}), log(new\ D(t', \overrightarrow{v})); e)) = trim(\hat{E}' new\ C(t, \overrightarrow{u}, log(new\ D(t', \overrightarrow{v})); e)).$$

Since $\tau'[\ldots \cdot n + 1] \approx \sigma' \kappa_1 \kappa_2 \kappa_3, \tau'[\ldots \cdot n + 1]$ is a prefix of $\tau$ and $L' \cap \{new\ D(t', \overrightarrow{v})\} \subseteq L$ due to the monotonic growth of log, we conclude that $new\ D(t', \overrightarrow{v}) \in L$.

Proof of Theorem 5.3.

Proof. Let $p$ be a source program and $LS$ be a logging specification defined as $LS = spec(X, \{MaybeBAD\})$. We aim to show that for all $R(p(\theta))$ and finite traces $\tau$ and $\sigma$, such that $R(p(\theta)) \vdash \sigma, \tau \vdash \sigma$ and $\sigma \vdash L$, $C[toFOL(L)] = LS(\tau)$. By Lemma B.17, we have

$$toFOL(L) = \delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)}.$$  

By Lemma B.10 and Lemma B.11

$$LS(\tau) = C(C(\delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)})) = C(\delta(X \cup toFOL(\tau)) \cap L_{(MaybeBAD)})$$.

Hence, $LS(\tau) \leq C(toFOL(L))$ and $C(toFOL(L)) \leq LS(\tau)$ both hold. \hfill \Box
p(new String("hello"))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(new String("hello").concat(new String("world"))))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(String.concat(new String(new String("hello"), val, new String("world"), val))))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(String.concat(new String(new String("hello"), new String("world"), val))))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(new String("hello world")))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(new String("hello world")))
  → TopLevel.main(new Sec().secureMeth(new Sec()).sanitize(new String("hello world")))
  → TopLevel.main(new String("hello world"))
  → new String("hello world")

Figure 9: Example A.1: Source trace, shadow expressions and target trace