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 managements 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]. The 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 attacks[1]. When a web-based software receives and

PLAS'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2993600.2993610

[1]We 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 taint 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 $\mathtt{p}(\theta)$ in this security setting contain a low integrity data source θ, 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 `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 of $SSOs$ of the same form, and for brevity we will write $sso(\mathtt{e})$ for a method invocation `new o.m(e)` on some object `o` where $\mathtt{C.m} \in SSO$. As a sanity condition we require $SSOs \cap Sanitizers = \varnothing$. 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-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 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.

in OpenMRS (version 2.4, released 7/15/2016, and the preceding versions) to the OpenMRS developing community. We discuss one particular case here.

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 $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}$ range over class names, $\mathtt{x}$ range over variables, $\mathtt{f}$ range over field names, and $\mathtt{m}$ range over method names. *Values*, denoted $\mathtt{v}$ or $\mathtt{u}$, are objects, i.e. expressions of the form $\mathtt{new\ C}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$. 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 $\mathtt{C.m}(\mathtt{e})$. 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.

$$
\begin{array}{lcl}
\mathtt{L} & ::= & \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} \\
\mathtt{M} & ::= & \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid \mathtt{C.m}(\mathtt{e}) \\
\mathtt{E} & ::= & [\,] \mid \mathtt{E.f} \mid \mathtt{E.m}(\bar{\mathtt{e}}) \mid \mathtt{v.m}(\bar{\mathtt{v}}, \mathtt{E}, \bar{\mathtt{e}}') \mid \mathtt{new\ C}(\bar{\mathtt{v}}, \mathtt{E}, \bar{\mathtt{e}}') \mid \mathtt{C.m}(\mathtt{E})
\end{array}
$$

For brevity in this syntax, we use vector notations. Specifically we write $\bar{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$, similarly for $\bar{\mathtt{C}}$, $\bar{\mathtt{m}}$, $\bar{\mathtt{x}}$, $\bar{\mathtt{e}}$, etc., and we write $\bar{\mathtt{M}}$ as shorthand for $\mathtt{M}_1 \cdots \mathtt{M}_n$. We write the empty sequence as $\varnothing$, we use a comma as a sequence concatenation operator. If and only if $\mathtt{m}$ is one of the names in $\bar{\mathtt{m}}$, we write $\mathtt{m} \in \bar{\mathtt{m}}$. Vector notation is also used to abbreviate sequences of declarations; we let $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$ and $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$; denote $\mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n$ and $\mathtt{C}_1\ \mathtt{f}_1; \ldots; \mathtt{C}_n\ \mathtt{f}_n;$ respectively. The notation $\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};$ abbreviates $\mathtt{this.f}_1 = \mathtt{f}_1; \ldots; \mathtt{this.f}_n = \mathtt{f}_n;$. Sequences of names and declarations are assumed to contain no duplicate names.

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.

$$fields_{CT}(\mathtt{Object}) = \varnothing$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad fields_{CT}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}}{fields_{CT}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}, \bar{\mathtt{C}}\ \bar{\mathtt{f}}}$$

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

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

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).

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

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

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.

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

Field
$$\frac{fields_{CT}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \qquad \mathtt{f}_i \in \bar{\mathtt{f}}}{\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{f}_i \to \mathtt{v}_i}$$

Invoke
$$\frac{mbody_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}, \mathtt{e}}{\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}) \to \mathtt{C.m}(\mathtt{e}[\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}][\bar{\mathtt{u}}/\bar{\mathtt{x}}])}$$

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

We use $\to^*$ to denote the reflexive, transitive closure of $\to$. We will also use the notion of an *execution trace* τ to represent a series of *configurations* κ, where $\tau = \kappa_1 \ldots \kappa_n$ means that $\kappa_i \to \kappa_{i+1}$ for $0 < i < n$. In the case of FJ, configurations κ are just expressions $\mathtt{e}$. 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 $\mathtt{p}(\theta)$ 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 $\mathtt{p}(\theta) = \mathtt{new\ TopLevel}().\mathtt{main}(\theta)$, and we write $\mathtt{p}(\theta) \Downarrow \tau$ iff trace τ begins with the configuration $\mathtt{p}(\theta)$.

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 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 [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 only, and a *library method* is any method that operates on primitive objects, defined in a primitive class. We expect primitive objects to be object wrappers for primitive values (e.g., `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 primitive class names.

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.

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

For library methods we require that the body of any library method be of the form where `C` is a primitive class:

$$\mathtt{return\ new\ C}(\bar{\mathtt{e}}_1, \ldots, \bar{\mathtt{e}}_n)$$

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 `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:

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

Similarly, to define string concatenation, we define a concatenation operation @ on primitive strings:

$$@(s_1, s_2) \approx s_1 s_2$$

and we extend the definition of `String` in CT with the following method, where we assume all `String` objects maintain their primitive representation in a `val` field:

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

3. 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" $\circ$ 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:* `new C().m(new String`(s_1)`, new String`(s_2)`). The proper shadow is:*

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

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).

3.1 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 Φ (the algebra). A partial ordering is induced on Φ 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 $X = Y$, iff $X \leq Y$ and $Y \leq X$. We assume given a 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 $\lfloor \sigma \rfloor \leq \lfloor \tau \rfloor$.

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 LS range over *logging specifications*, which are functions from traces to Φ. Intuitively, $LS(\tau)$ denotes the information that should be recorded in an audit log during the execution of τ given specification LS. This is the semantics of the logging specification LS. Assuming also an interpretation of logs $\lfloor \cdot \rfloor$ 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* $\mathbb{L}$ *is* sound with respect to logging specification LS and execution trace τ *iff* $\lfloor \mathbb{L} \rfloor \leq LS(\tau)$. *Similarly, audit log* $\mathbb{L}$ *is* complete with respect to logging specification LS and execution trace τ *iff* $LS(\tau) \leq \lfloor \mathbb{L} \rfloor$.

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 ϕ and ψ 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* Φ_{FOL} *of closed sets of formulas:*

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

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

Let *Preds* be the set of all predicate symbols, and let $S \subseteq Preds$ 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 *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:* $X^{\Rightarrow S} = C(X \cap L_S)$ *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 $\lfloor \cdot \rfloor = C(toFOL(\cdot))$. Intuitively, in the interpretation each configuration is represented by a $\mathrm{Context}$ 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.

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

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

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

Figure 2: match **predicate definition.**

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

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

such that $toFOL(\sigma) = \bigcup_i toFOL(\kappa_i, i)$ *for* $\sigma = \kappa_1 \cdots \kappa_k$*. We define* $toFOL(\kappa, n)$ *as in Figure 1.*

3.2.1 Integrity Identifiers

We introduce an integrity identifier t that denotes the integrity level associate 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 questionable integrity ($\odot$).

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

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 $\wedge$ 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 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 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 objects:

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

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

DEFINITION 3.5. *Shadow method bodies are defined by the function* $smbody$.

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

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

$$\begin{array}{l} srewrite(\texttt{x}) = \texttt{x} \\ srewrite(\texttt{new C}(\bar{\texttt{e}})) = \texttt{shadow C}(\circ, srewrite(\bar{\texttt{e}})) \\ srewrite(\texttt{e.f}) = srewrite(\texttt{e}).\texttt{f} \\ srewrite(\texttt{e.m}(\bar{\texttt{e}}')) = srewrite(\texttt{e}).\texttt{m}(srewrite(\bar{\texttt{e}}')) \\ srewrite(\texttt{C.m}(\texttt{e})) = \texttt{C.m}(srewrite(\texttt{e})) \\ srewrite(Op(\bar{\texttt{e}})) = Op(srewrite(\bar{\texttt{e}})) \\ srewrite(\nu) = \delta \end{array}$$

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 $\mathrm{Shadow}(n, se)$ 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.[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 $\mathrm{Prop}(t, T)$ where T is a compound term of the form $\texttt{C.m}(\bar{t})$ with $\bar{t}$ the given integrity of this followed by the integrity of the arguments to method C.m, and t is the integrity of the result. For example, one could define:

$$\mathrm{meet}(t, t_1, t_2) \Rightarrow \mathrm{Prop}(t, \texttt{Int.plus}(t_1, t_2))$$

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

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 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 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* Prop*. The prospective integrity taint analysis policy is defined as the set of traces that do not end in* BAD *states.*

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

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

$$LS_{\mathrm{taint}} = \lambda\tau.(\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\mathrm{MaybeBAD}\}}$$

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

DEFINITION 3.7. *We call a program* $\mathtt{p}(\theta)$ safe *iff for all* τ *it is the case that* $\mathtt{p}(\theta) \Downarrow \tau$ *implies* $\tau \in \mathrm{SP}_{\mathrm{taint}}$*. We call the program* unsafe *iff there exists some trace* τ *such that* $\mathtt{p}(\theta) \Downarrow \tau$ *and* $\tau \notin \mathrm{SP}_{\mathrm{taint}}$.

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

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

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

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

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

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

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

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

$\text{Shadow}(n, se) \wedge \text{match}(se, SE, \mathit{Op}(\bar{\delta})) \implies \text{Shadow}(n, SE[\delta]).$

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

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

Figure 3: Shadow **predicate definition.**

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

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

Figure 4: Predicates for Specifying Prospective and Retrospective Properties

blocks if the argument is tainted, an `endorse` method for any object class returned by a sanitizer, and modification of all sanitizers to `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 `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 FJ_{taint} below.

DEFINITION 4.1. *For any expression* $\texttt{e}$*, the expression* $\mu(\texttt{e})$ *is syntactically equivalent to* $\texttt{e}$ *except with every subexpression* $\texttt{new C}(\bar{\texttt{e}})$ *replaced with* $\texttt{new C}(\circ, \bar{\texttt{e}})$*. Given SSOs and Sanitizers, define* $\mathcal{R}(\texttt{e}, CT) = (\mu(\texttt{e}), \mathcal{R}(CT))$*, where* $\mathcal{R}(CT)$ *is the smallest class table satisfying the axioms given in Figure 5.*

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 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:

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

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

4.3 Operational Semantics of FJ_{taint}

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

$$\text{Reduce}\ \frac{\texttt{e} \to \texttt{e}'}{\texttt{e}, \mathbb{L} \to \texttt{e}', \mathbb{L}} \qquad \text{Sequence}\ \ \texttt{v}; \texttt{e} \to \texttt{e}$$

$$\text{Log}\ \frac{t \leq \odot}{\texttt{u.log}(\texttt{new C}(t, \bar{\texttt{v}})), \mathbb{L} \to \texttt{new C}(t, \bar{\texttt{v}}), (\mathbb{L}, \texttt{new C}(t, \bar{\texttt{v}}))}$$

$$\text{NoLog}\ \frac{t > \odot}{\texttt{u.log}(\texttt{new C}(t, \bar{\texttt{v}})), \mathbb{L} \to \texttt{new C}(t, \bar{\texttt{v}}), \mathbb{L}}$$

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

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.

EXAMPLE 4.1. *Let*

$\mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel}) =$
$\texttt{x}, \texttt{new Sec}().\texttt{secureMeth}(\texttt{new Sec}().\texttt{sanitize}(\texttt{x.concat}($
$\texttt{new String}(\texttt{"world"}))).$

Assume the string $\texttt{"hello"}$ *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*

$$fields_{\mathcal{R}(CT)}(\texttt{Object}) = \texttt{Taint taint} \qquad mbody_{\mathcal{R}(CT)}(\texttt{check}, \texttt{Object}) = \texttt{x}, \texttt{new Object}(?\texttt{x.taint})$$

$$\frac{\texttt{C.m} \in Sanitizers \qquad mtype_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{C}} \to \texttt{D} \qquad fields_{CT}(\texttt{D}) = \bar{\texttt{f}}}{mbody_{\mathcal{R}(CT)}(\texttt{endorse}, \texttt{D}) \quad = \quad \varnothing, \texttt{new D}(\vee(\odot, \texttt{this.taint}), \overline{\texttt{this.f}})}$$

$$\frac{\texttt{C.m} \in SSOs \qquad mbody_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{e}}{mbody_{\mathcal{R}(CT)}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{this.log(x)}; \texttt{this.check(x)}; \mu(\texttt{e})}$$

$$\frac{\texttt{C.m} \notin Sanitizers \cup SSOs \qquad mbody_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}}{mbody_{\mathcal{R}(CT)}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \mu(\texttt{e})}$$

$$\frac{\texttt{C.m} \in Sanitizers \qquad mbody_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}}{mbody_{\mathcal{R}(CT)}(\_\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \mu(\texttt{e})}$$

$$\frac{\texttt{C.m} \in Sanitizers \qquad mbody_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}}{mbody_{\mathcal{R}(CT)}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{this.\_m}(\bar{\texttt{x}})\texttt{.endorse}()}$$

Figure 5: Axioms for Rewriting Algorithm

methods Sec.sanitize *and* Sec.secureMeth *are identity functions. Some reduction steps are elided in the example as n-length multi-step reductions $\to^n$. 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 $\mathrm{SP}_{\mathrm{taint}}$ 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 $\mathfrak{p}(\theta)$ (Definition 3.7) all of the following hold:*

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

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 $\mathfrak{p}(\theta)$ with rewritten programs $\mathcal{R}(\mathfrak{p}(\theta))$ 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 $\mathfrak{p}$ and class tables CT once, prior to invocation on possibly tainted θ, due to the following property which follows immediately from the definition of $\mathcal{R}$.

LEMMA 5.1. $\mathcal{R}(\mathfrak{p}(\theta)) = \mathcal{R}(\mathfrak{p})(\mathcal{R}(\theta))$

5.2 Correctness for Prospective Analysis

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

DEFINITION 5.2. *An $\mathrm{FJ}_{\mathrm{taint}}$ program* e causes a security failure *iff* $\texttt{e}, \varnothing \to^* \texttt{E}[\texttt{v.check(new C}(\bullet, \bar{\texttt{v}}))], \mathbb{L}$ *for some* E, v, $\texttt{new C}(\bullet, \bar{\texttt{v}})$, *and* $\mathbb{L}$.

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 $\mathfrak{p}(\theta)$ is unsafe (Definition 3.7) iff $\mathcal{R}(\mathfrak{p}(\theta))$ 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 τ describes execution of a source program, rewriting should produce a program with a trace τ' that corresponds to τ (i.e., $\tau :\approx \tau'$), where the log $\mathbb{L}$ generated by τ', written $\tau' \rightsquigarrow \mathbb{L}$, 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 $\mathrm{FJ}_{\mathrm{taint}}$ programs we write $\tau \rightsquigarrow \mathbb{L}$ iff $tail(\sigma) = \texttt{e}, \mathbb{L}$ where σ is the longest trace such that $\tau' :\approx \tau$ and $\tau' :\approx \sigma$ for some FJ trace τ'.*

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 $toFOL(\mathbb{L}) = \{\mathrm{MaybeBAD}(\texttt{v}) \mid \texttt{v} \in \mathbb{L}\}$, and thus $\lfloor \mathbb{L} \rfloor = C(toFOL(\mathbb{L}))$.

DEFINITION 5.5. *Rewriting algorithm $\mathcal{R}$ is* retrospectively sound/-complete *iff for all programs $\mathcal{R}(\mathfrak{p}(\theta))$, and finite traces τ and σ where:*

$$\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \sigma \qquad \tau :\approx \sigma \qquad \sigma \rightsquigarrow \mathbb{L}$$

we have that $\mathbb{L}$ is sound/complete with respect to LS_{taint} and τ.

$$\texttt{p}(\texttt{new String}(''\texttt{hello }'')) \rightarrow^5 \texttt{TopLevel.main}(\texttt{new Sec}().\texttt{secureMeth}(\texttt{new Sec}().\texttt{sanitize}(\texttt{new String}(''\texttt{hello world}''))))$$
$$\rightarrow^2 \texttt{TopLevel.main}(\texttt{new Sec}().\texttt{secureMeth}(\texttt{new String}(''\texttt{hello world}''))) \rightarrow^3 \texttt{new String}(''\texttt{hello world}'').$$

$$\text{Shadow}(1, \texttt{shadow TopLevel}(\circ).\texttt{main}(\texttt{shadow String}(\bullet)))$$
$$\text{Shadow}(6, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ).\texttt{secureMeth}(\texttt{shadow Sec}(\circ).\texttt{sanitize}(\texttt{shadow String}(\bullet)))))$$
$$\text{Shadow}(8, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ).\texttt{secureMeth}(\texttt{shadow String}(\odot)))) \qquad \text{Shadow}(11, \texttt{shadow String}(\odot))$$

$$\mathcal{R}(\texttt{p}(\texttt{new String}(''\texttt{hello }'')), \varnothing \rightarrow^5 \texttt{TopLevel.main}(\texttt{new Sec}(\circ).\texttt{secureMeth}(\texttt{new Sec}(\circ).\texttt{sanitize}(\texttt{new String}(\bullet, ''\texttt{hello world}''))))), \varnothing$$
$$\rightarrow \texttt{TopLevel.main}(\texttt{new Sec}(\circ).\texttt{secureMeth}(\texttt{Sec.sanitize}(\texttt{new Sec}(\circ).\_\texttt{sanitize}(\texttt{new String}(\bullet, ''\texttt{hello world}'')).\texttt{endorse}()))), \varnothing$$
$$\rightarrow^3 \texttt{TopLevel.main}(\texttt{new Sec}(\circ).\texttt{secureMeth}(\texttt{new String}(\odot, ''\texttt{hello world}''))), \varnothing$$
$$\rightarrow \texttt{TopLevel.main}(\texttt{Sec.secureMeth}(\texttt{new Sec}(\circ).\texttt{log}(\texttt{new String}(\odot, ''\texttt{hello world}'')); \texttt{new Sec}(\circ).\texttt{check}(\texttt{new String}(\odot, ''\texttt{hello world}'')); \texttt{new String}(\odot, ''\texttt{hello world}''))), \varnothing \rightarrow^4 \texttt{new String}(\odot, ''\texttt{hello world}''), \{\texttt{new String}(\odot, ''\texttt{hello world}'')\}$$

Figure 6: Example 4.1: Source trace, shadow expressions and target trace

$$overlay(\texttt{x}, \texttt{x}) = \texttt{x} \qquad overlay(\nu, \delta) = \nu$$
$$overlay(Op(\bar{\texttt{e}}), Op(\overline{se})) = Op(\overline{overlay(\texttt{e}, se)})$$
$$overlay(\texttt{e.f}, se.\texttt{f}) = overlay(\texttt{e}, se).\texttt{f}$$
$$overlay(\texttt{new C}(\bar{\texttt{e}}), \texttt{shadow C}(t, \overline{se})) = \texttt{new C}(t, \overline{overlay(\texttt{e}, se)})$$
$$overlay(\texttt{C.m}(\texttt{e}), \texttt{C.m}(se)) = \texttt{C.m}(overlay(\texttt{e}, se))$$
$$overlay(\texttt{e.m}(\bar{\texttt{e}'}), se.\texttt{m}(\overline{se'})) = overlay(\texttt{e}, se).\texttt{m}(\overline{overlay(\texttt{e}', se')})$$

Figure 7: Definition of $overlay$.

$$trim(\texttt{x}) = \texttt{x} \qquad trim(\texttt{e.f}) = trim(\texttt{e}).\texttt{f}$$
$$trim(\texttt{new C}(\bar{\texttt{e}})) = \texttt{new C}(\overline{trim(\texttt{e})}) \qquad trim(\texttt{C.m}(\texttt{e})) = \texttt{C.m}(trim(\texttt{e}))$$
$$trim(Op(\bar{e})) = Op(\overline{trim(e)})$$
$$trim(\texttt{e}_1; \texttt{e}_2) = trim(\texttt{e}_1); trim(\texttt{e}_2) \qquad trim(\texttt{e.m}(\bar{\texttt{e}'})) =$$
$$\begin{cases} trim(\texttt{e}) & \text{if } \texttt{m} = \texttt{endorse} \\ \epsilon & \text{if } \texttt{m} \in \{\texttt{log}, \texttt{check}\} \\ trim(\texttt{e}).\texttt{m}(\overline{trim(\texttt{e}')}) & \text{if } \texttt{m} \notin \{\texttt{log}, \texttt{check}, \texttt{endorse}\} \end{cases}$$

Figure 8: Definition of $trim$.

5.4 Definition of $:\approx$ and Correctness Results

To establish correctness of program rewriting, we need to define a correspondence relation $:\approx$. Source language execution traces and target language execution traces correspond if they represent the same expression evaluated to the same point. We make 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* : $(\texttt{e}, se) \mapsto \texttt{e}'$ *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, $LastShadow(\tau)$ denotes the shadow of the last configuration in the trace τ. Considering the rule

$$\text{Shadow}(n, se) \implies \text{LShadow}(se), \tag{1}$$

we define $LastShadow(\tau) = se$ such that $\lfloor \tau \rfloor \otimes X \vdash \text{LShadow}(se)$, where X contains the rules given in Figure 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* : $\texttt{e} \mapsto \texttt{e}'$ *is defined in Figure 8. We assume ϵ to be no-op, i.e., $\epsilon; \texttt{e} = \texttt{e}$.*

DEFINITION 5.8. *Given source language execution trace $\tau = \sigma\kappa$ and target language execution trace $\tau' = \sigma'\kappa'$, $\tau :\approx \tau'$ iff $overlay(\kappa, LastShadow(\tau)) = trim(\texttt{e}')$, where $\kappa' = \texttt{e}', \mathbb{L}$.*

Theorem 5.1 establishes semantics preservation for rewriting algorithm $\mathcal{R}$. Moreover, Theorem 5.2 and Theorem 5.3 establish prospective and retrospective correctness of $\mathcal{R}$ respectively. Proofs for these theorems are given in Section B.

THEOREM 5.1. *The rewriting algorithm $\mathcal{R}$ is semantics preserving (Definition 5.1).*

THEOREM 5.2. *The rewriting algorithm $\mathcal{R}$ is prospectively correct (Definition 5.3).*

THEOREM 5.3. *The rewriting algorithm $\mathcal{R}$ 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. Haldar 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

[1] S. Amir-Mohammadian, S. Chong, and C. Skalka. Correct audit logging: Theory and practice. In *POST*, pages 139–162, 2016.

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

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

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

[5] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *RAID*, pages 1–20, 2011.

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

[7] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *IEEE ISCC*, pages 749–754, 2006.

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

[9] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.

[10] V. Ganapathy, T. Jaeger, C. Skalka, and G. Tan. Assurance for defense in depth via retrofitting. In *LAW*, 2014.

[11] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *ACSAC*, pages 303–311, 2005.

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

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

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

[15] B. Livshits, M. Martin, and M. S. Lam. Securifly: Runtime protection and recovery from web application vulnerabilities. Technical report, Technical report, Stanford University, 2006.

[16] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: A program query language. In *OOPSLA*, 2005.

[17] U. Nilsson and J. Maluszyynski. Definite logic programs. In *Logic, Programming and Prolog*, chapter 2. 2000.

[18] OpenMRS. http://openmrs.org/, 2016. Accessed: 2016-07-28.

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

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

[21] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE EuroS&P*, pages 15–30, 2016.

[22] E. J. Schwartz, T. Avgerinos, and D. 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.

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

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

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

[26] Z. Wei and D. Lie. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *SPSM@CCS*, pages 27–38, 2014.

[27] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *Operating Systems Review*, 45(1):142–154, 2011.

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*

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

and assume the string $''\texttt{hello}''$ *is tainted with low integrity. Figure 9 gives the FJ trace, the shadow expressions and the* FJ_{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 $\texttt{e}$ is reduced to $\texttt{e}'$ then $\texttt{e}$ reduces (in potentially multiple steps) to some expression with the same $trim$ as $\texttt{e}'$.

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

PROOF. By induction on the structure of $\texttt{e}$. □

LEMMA B.2. *For all expressions* $\mathtt{e}$*, if* $trim(\mathtt{e}), \mathbb{L} \rightarrow \mathtt{e}', \mathbb{L}'$ *then either (1) there exists* σ *such that* $\mathtt{e}, \mathbb{L} \Downarrow \sigma\kappa$ *with* $\kappa = \mathtt{e}'', \mathbb{L}''$ *and* $trim(\mathtt{e}'') = trim(\mathtt{e}')$*, or (2)* $\mathtt{e}$ *causes a security failure.*

PROOF. By induction on the structure of $\mathtt{e}$, and applying Lemma B.1. □

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

LEMMA B.3. $overlay(\mathtt{e}, srewrite(\mathtt{e})) = \mu(\mathtt{e})$.

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

Lemma B.4 and Lemma B.5 state that single step and multi step reductions in FJ preserve $:\approx$.

LEMMA B.4. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathtt{e}_1 \rightarrow \mathtt{e}_1'$ *then there exists* σ *such that* $\kappa_2 \Downarrow \sigma$ *and* $\tau_1\mathtt{e}_1\mathtt{e}_1' :\approx \tau_2\sigma$.

PROOF. By induction on the derivation of $\mathtt{e}_1 \rightarrow \mathtt{e}_1'$ and applying Lemmas B.2 and B.3. □

LEMMA B.5. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathtt{e}_1 \Downarrow \sigma_1$*, then there exists* σ_2 *such that* $\kappa_2 \Downarrow \sigma_2$ *and* $\tau_1\sigma_1 :\approx \tau_2\sigma_2$.

PROOF. By induction on the derivation of $\mathtt{e}_1 \Downarrow \sigma_1$ and applying Lemma B.4. □

Similarly, Lemma B.6 and Lemma B.7 argue that single step and multi step reductions in FJ_{taint} preserve $:\approx$.

LEMMA B.6. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\kappa_2 \rightarrow \kappa_2'$ *then there exists* σ *where* $\mathtt{e}_1 \Downarrow \sigma$ *and* $\tau_1\sigma :\approx \tau_1\kappa_2\kappa_2'$.

PROOF. By induction on the derivation of $\kappa_2 \rightarrow \kappa_2'$ and applying Lemma B.3. □

LEMMA B.7. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\kappa_2 \Downarrow \sigma_2$*, then there exists* σ_1 *such that* $\mathtt{e}_1 \Downarrow \sigma_1$ *and* $\tau_1\sigma_1 :\approx \tau_2\sigma_2$.

PROOF. By induction on the derivation of $\kappa_2 \Downarrow \sigma_2$ and applying Lemma B.6. □

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

LEMMA B.8. *Let* $\theta = \mathtt{new\ C}(\bar{\nu})$ *be a tainted input. Then,*

$$\mathtt{new\ TopLevel}().\mathtt{main}(\theta) :\approx \mathtt{new\ TopLevel}(\circ).\mathtt{main}(\mathtt{new\ C}(\bullet, \bar{\nu})), \varnothing.$$

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

Proof of Theorem 5.1.

PROOF. Lemma B.8 states that initial configuration of a program $\mathtt{p}$ corresponds to the initial configuration of $\mathcal{R}(\mathtt{p})$. 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 SP_{taint} is a safety property.

LEMMA B.9. SP_{taint} *is a safety property.*

PROOF. Let $\tau \notin \text{SP}_{\text{taint}}$. Then, $(\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\text{BAD}\}} \neq C(\varnothing)$. This implies that there exists some n such that $\text{BAD}(n) \in (\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\text{BAD}\}}$. Let $\tau[\cdots n]$ 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 $\lfloor\tau\rfloor \otimes C(X) \vdash \text{BAD}(n)$ iff $\lfloor\tau[\cdots n]\rfloor \otimes C(X) \vdash \text{BAD}(n)$, i.e. $\tau \notin \text{SP}_{\text{taint}}$ iff $\tau[\cdots n] \notin \text{SP}_{\text{taint}}$ for finite n, hence SP_{taint} is a safety property [20]. □

Proof of Theorem 5.2.

PROOF. Suppose on the one hand that $\mathtt{p}(\theta)$ is unsafe, which is the case iff $\mathtt{p}(\theta) \Downarrow \tau$ and $\tau \notin \text{SP}_{\text{taint}}$ for some τ. Then according to Lemma B.9, there exists some timestamp n such that $\tau[\cdots n]$ characterizes SP_{taint}, i.e., $\text{BAD}(n)$ is derivable from the rules in Definition 3.6 and so $(\tau[\cdots n])\sigma \notin \text{SP}_{\text{taint}}$ for any σ. Let n be the least timestamp with such property and $\tau' = \tau[\cdots n-1]$, and $\tau[n] = \mathtt{E}[\mathtt{v.m(new\ D}(\bar{\mathtt{u}}))]$ where $\mathtt{v.m}$ is an sso invocation and $\mathtt{new\ D}(\bar{\mathtt{u}})$ has low integrity by the Definition 3.6. By Theorem 5.1 there exists some trace σ, such that $\mathcal{R}(\mathtt{p}(\theta)) \Downarrow \sigma$ and $\tau' :\approx \sigma$. Therefore, by definition of $:\approx$ and $\rightarrow$ we may assert that $tail(\sigma) \Downarrow \kappa_1\kappa_2\kappa_3$ such that

$$\kappa_1 = \mathtt{E}'[\mathtt{v.m(new\ D}(\bullet, \bar{\mathtt{u}}))], \mathbb{L}$$
$$\kappa_2 = \mathtt{E}'[\mathtt{v.log(new\ D}(\bullet, \bar{\mathtt{u}})); \mathtt{v.check(new\ D}(\bullet, \bar{\mathtt{u}}))], \mathbb{L}$$
$$\kappa_3 = \mathtt{E}'[\mathtt{v.check(new\ D}(\bullet, \bar{\mathtt{u}}))], \mathbb{L} \cup \{\mathtt{new\ D}(\bullet, \bar{\mathtt{u}})\}$$

Thus, $\mathcal{R}(\mathtt{p}(\theta)) \Downarrow \sigma\kappa_1\kappa_2\kappa_3$ and therefore $\mathcal{R}(\mathtt{p}(\theta))$ causes a security failure.

Supposing on the other hand that $\mathcal{R}(\mathtt{p}(\theta))$ causes a security failure, it follows that $\mathtt{p}(\theta)$ is unsafe by similar reasoning (i.e. fundamental appeal to Theorem 5.1), since security `checks` are only added to the beginning of *SSOs* 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 $\mathfrak{H}$ 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 FJ_{taint} 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. $C(\mathfrak{H}(X)) = C(X)$ *and* $\mathfrak{H}(X) = \mathfrak{H}(C(X))$.

LEMMA B.11. $C(C(\mathfrak{H}(X)) \cap L) = C(\mathfrak{H}(X) \cap L)$.

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* $\mathtt{e}, \mathbb{L} \to \mathtt{e}', \mathbb{L}'$. *If* $\mathtt{v} \in \mathbb{L}' - \mathbb{L}$ *then we have* $\mathtt{e} = \mathtt{E}[\mathtt{u.log(v)}]$ *for some evaluation context* E *and value* u, $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ *and* $t \leq \odot$.

PROOF. By induction on the derivation of $\mathtt{e}, \mathbb{L} \to \mathtt{e}', \mathbb{L}'$. □

LEMMA B.13. *Let* $\mathtt{e}, \mathbb{L} \Downarrow \sigma\kappa$, *where* $\kappa = \mathtt{e}', \mathbb{L}'$. *If* $\mathtt{v} \in \mathbb{L}' - \mathbb{L}$ *then there exists some trace* $\sigma'\kappa_1\kappa_2$ *as the prefix of* $\sigma\kappa$ *such that* $\kappa_1 = \mathtt{e}_1, \mathbb{L}_1$, $\kappa_2 = \mathtt{e}_2, \mathbb{L}_2$, *where* $\mathtt{e}_1 = \mathtt{E}[\mathtt{u.log(v)}]$ *for some evaluation context* E *and value* u, $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ *and* $t \leq \odot$.

PROOF. By induction on the derivation of $\mathtt{e}, \mathbb{L} \Downarrow \sigma\kappa$ and applying Lemma B.12. □

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

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

PROOF. By induction on the structure of e, and applying Lemma B.1. □

LEMMA B.15. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathtt{e}_1 \to \mathtt{e}_1'$ *then there exists* σ *such that* $\kappa_2 \Downarrow \sigma$, $\tau_1\mathtt{e}_1\mathtt{e}_1' :\approx \tau_2\sigma$, *and if* $tail(\sigma) = \kappa_2'$ *and* $\kappa_2' \to \kappa_2''$ *for some* κ_2'' *then* $\tau_1\mathtt{e}_1\mathtt{e}_1' :\not\approx \tau_2\sigma\kappa_2''$.

PROOF. By induction on the derivation of $\mathtt{e}_1 \to \mathtt{e}_1'$ and applying Lemmas B.14 and B.3. □

LEMMA B.16. *If* $\tau_1\mathtt{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathtt{e}_1 \Downarrow \sigma_1$ *non-trivially, then there exists* σ_2 *such that* $\kappa_2 \Downarrow \sigma_2$, $\tau_1\sigma_1 :\approx \tau_2\sigma_2$, *and if* $tail(\sigma_2) = \kappa_2'$ *and* $\kappa_2' \to \kappa_2''$ *for some* κ_2'' *then* $\tau_1\sigma_1 :\not\approx \tau_2\sigma_2\kappa_2''$.

PROOF. By induction on the derivation of $\mathtt{e}_1 \Downarrow \sigma_1$ and applying Lemma B.15. □

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 retrospective correctness in Theorem 5.3.

LEMMA B.17. *Given* $\mathcal{R}(\mathtt{p}(\theta)) \Downarrow \sigma$ *and* $\tau :\approx \sigma$ *and* $\sigma \rightsquigarrow \mathbb{L}$, *we have:*

$$toFOL(\mathbb{L}) = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}.$$

PROOF. (Sketch) We first show that $toFOL(\mathbb{L})$ is a subset of $\mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}$. Let $\text{MaybeBAD}(\mathtt{v}) \in toFOL(\mathbb{L})$. According to the definition of $toFOL(\mathbb{L})$, $\mathtt{v} \in \mathbb{L}$. Using Lemma B.13, there exists some trace $\sigma'\kappa_1\kappa_2$ as the prefix of σ such that $\kappa_1 = \mathtt{e}_1, \mathbb{L}_1$, $\kappa_2 = \mathtt{e}_2, \mathbb{L}_2$, where $\mathtt{e}_1 = \mathtt{E}[\mathtt{u.log(v)}]$ for some evaluation context E and value u, $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ and $t \leq \odot$. Using Theorem 5.1, there exists a trace $\hat{\tau}$ such that $\mathtt{p}(\theta) \Downarrow \hat{\tau}$ and $\hat{\tau} :\approx \sigma'\kappa_1\kappa_2$. This ensures that

$$\text{MaybeBAD}(\mathtt{v}) \in \mathfrak{H}(X \cup toFOL(\hat{\tau})) \cap L_{\{\text{MaybeBAD}\}},$$

as $\mathtt{u.log(v)}$ could only appear in the body of some method $\mathtt{C.m} \in SSOs$, according to the rewriting algorithm $\mathcal{R}$, and thus the preconditions of the last rule defined in Figure 4 are satisfied by $X \cup toFOL(\hat{\tau})$. Moreover, $\hat{\tau}$ is a prefix of τ, and thus $toFOL(\hat{\tau}) \subseteq toFOL(\tau)$. This entails that

$$\text{MaybeBAD}(\mathtt{v}) \in \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}.$$

Next, we show that $\mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}$ is a subset of $toFOL(\mathbb{L})$. Let $\text{MaybeBAD}(\mathtt{new\ D}(t', \overline{\mathtt{v}'})) \in \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}$ and $\mathtt{v} = \mathtt{new\ D}(\overline{\mathtt{v}'})$. Then, there exist some n, C, $\bar{\mathtt{u}}$ and m where $\text{Call}(n, \mathtt{C}, \bar{\mathtt{u}}, \mathtt{m}, \mathtt{v}) \in toFOL(\tau)$. Moreover, there exist SE, se, t, $\overline{se}$ and $\overline{se'}$, where $\text{Shadow}(n, se)$ and $\text{match}(se, SE, \mathtt{shadow\ C}(t, \overline{se}).\mathtt{m}(\mathtt{shadow\ D}(t', \overline{se'})))$ are derivable from the rules in X, $\mathtt{C.m} \in SSOs$ and $t' \leq \odot$. Let $\tau[\cdots n]$ denote the prefix of τ ending in timestamp n. Based on the definition of $toFOL(\cdot)$, we can infer that $tail(\tau[\cdots n]) = \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{u}}).\mathtt{m}(\mathtt{v})]$. Using Theorem 5.1, we know that there exists trace σ' such that $\mathcal{R}(\mathtt{p}(\theta)) \Downarrow \sigma'$ and $\tau[\cdots n] :\approx \sigma'$. Let $tail(\sigma') = \hat{\mathtt{e}}, \hat{\mathbb{L}}$. Therefore,

$$\begin{aligned} trim(\hat{\mathtt{e}}) &= overlay(\mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{u}}).\mathtt{m}(\mathtt{new\ D}(\overline{\mathtt{v}'}))], \\ &\qquad SE[\mathtt{shadow\ C}(t, \overline{se}).\mathtt{m}(\mathtt{shadow\ D}(t', \overline{se'}))]) \\ &= \hat{\mathtt{E}}[\mathtt{new\ C}(t, \bar{\mathtt{u}}).m(\mathtt{new\ D}(t', \bar{\mathtt{v}}'))]. \end{aligned}$$

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

$$\begin{aligned} \kappa_1 &= \hat{\mathtt{E}}'[\mathtt{new\ C}(t, \bar{\mathtt{u}}).m(\mathtt{new\ D}(t', \bar{\mathtt{v}}'))], \hat{\mathbb{L}}' \\ \kappa_2 &= \hat{\mathtt{E}}'[\mathtt{new\ C}(t, \bar{\mathtt{u}}).\mathtt{log}(\mathtt{new\ D}(t', \bar{\mathtt{v}}'))\mathtt{;\ e}], \hat{\mathbb{L}}' \\ \kappa_3 &= \hat{\mathtt{E}}'[\mathtt{e}], \hat{\mathbb{L}}' \cup \{\mathtt{new\ D}(t', \bar{\mathtt{v}}')\}, \end{aligned}$$

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

$$\begin{aligned} &trim(\hat{\mathtt{E}}'[\mathtt{new\ C}(t, \bar{\mathtt{u}}).\mathtt{log}(\mathtt{new\ D}(t', \bar{\mathtt{v}}'))\mathtt{;\ e}]) = \\ &\qquad trim(\hat{\mathtt{E}}[\mathtt{new\ C}(t, \bar{\mathtt{u}}).\mathtt{log}(\mathtt{new\ D}(t', \bar{\mathtt{v}}'))\mathtt{;\ e}]). \end{aligned}$$

Since $\tau[\cdots n+1] :\approx \sigma'\sigma''\kappa_1\kappa_2\kappa_3$, $\tau[\cdots n+1]$ is a prefix of τ and $\hat{\mathbb{L}}' \cup \{\mathtt{new\ D}(t', \bar{\mathtt{v}}')\} \subseteq \mathbb{L}$ due to the monotonic growth of log, we conclude that $\mathtt{new\ D}(t', \bar{\mathtt{v}}') \in \mathbb{L}$. □

Proof of Theorem 5.3.

PROOF. Let p be a source program and LS be a logging specification defined as $LS = spec(X, \{\text{MaybeBAD}\})$. We aim to show that for all $\mathcal{R}(\mathtt{p}(\theta))$ and finite traces τ and σ, such that $\mathcal{R}(\mathtt{p}(\theta)) \Downarrow \sigma, \tau :\approx \sigma$ and $\sigma \rightsquigarrow \mathbb{L}$, $C(toFOL(\mathbb{L})) = LS(\tau)$. By Lemma B.17, we have

$$toFOL(\mathbb{L}) = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}.$$

By Lemma B.10 and Lemma B.11

$$\begin{aligned} LS(\tau) &= C(C(\mathfrak{H}(X \cup toFOL(\tau))) \cap L_{\{\text{MaybeBAD}\}}) \\ &= C(\mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{MaybeBAD}\}}). \end{aligned}$$

Hence, $LS(\tau) \leq C(toFOL(\mathbb{L}))$ and $C(toFOL(\mathbb{L})) \leq LS(\tau)$ both hold. □

$\mathtt{p}(\mathtt{new\ String}(''\mathtt{hello\ }''))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ Sec}().\mathtt{sanitize}(\mathtt{new\ String}(''\mathtt{hello\ }'').\mathtt{concat}(\mathtt{new\ String}(''\mathtt{world}'')))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ Sec}().\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{new\ String}(@(\mathtt{new\ String}(''\mathtt{hello\ }'').\mathtt{val},$
$\mathtt{new\ String}(''\mathtt{world}'').\mathtt{val}))))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ Sec}().\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{new\ String}(@(''\mathtt{hello\ }'', \mathtt{new\ String}(''\mathtt{world}'').\mathtt{val}))))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ Sec}().\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{new\ String}(''\mathtt{hello\ world}'')))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ Sec}().\mathtt{sanitize}(\mathtt{new\ String}(''\mathtt{hello\ world}''))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{Sec.sanitize}(\mathtt{new\ String}(''\mathtt{hello\ world}''))))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}().\mathtt{secureMeth}(\mathtt{new\ String}(''\mathtt{hello\ world}'')))$

$\to \mathtt{TopLevel.main}(\mathtt{Sec.secureMeth}(\mathtt{new\ String}(''\mathtt{hello\ world}'')))$

$\to \mathtt{TopLevel.main}(\mathtt{new\ String}(''\mathtt{hello\ world}''))$

$\to \mathtt{new\ String}(''\mathtt{hello\ world}'').$

$\mathrm{Shadow}(1, \mathtt{shadow\ TopLevel}(\circ).\mathtt{main}(\mathtt{shadow\ String}(\bullet, \delta)))$

$\mathrm{Shadow}(2, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}(\mathtt{shadow\ String}(\bullet, \delta).\mathtt{concat}(\mathtt{shadow\ String}(\circ, \delta))))))$

$\mathrm{Shadow}(3, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}($
$\mathtt{String.concat}(\mathtt{shadow\ String}(\bullet, @(\mathtt{shadow\ String}(\bullet, \delta).\mathtt{val}, \mathtt{shadow\ String}(\circ, \delta).\mathtt{val})))))))$

$\mathrm{Shadow}(4, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}($
$\mathtt{String.concat}(\mathtt{shadow\ String}(\bullet, @(\delta, \mathtt{shadow\ String}(\circ, \delta).\mathtt{val})))))))$

$\mathrm{Shadow}(5, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{shadow\ String}(\bullet, @(\delta, \delta)))))))$

$\mathrm{Shadow}(5, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{shadow\ String}(\bullet, \delta))))))$

$\mathrm{Shadow}(6, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ Sec}(\circ).\mathtt{sanitize}(\mathtt{shadow\ String}(\bullet, \delta)))))$

$\mathrm{Shadow}(7, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{Sec.sanitize}(\mathtt{shadow\ String}(\bullet, \delta)))))$

$\mathrm{Shadow}(8, \mathtt{TopLevel.main}(\mathtt{shadow\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{shadow\ String}(\odot, \delta))))$

$\mathrm{Shadow}(9, \mathtt{TopLevel.main}(\mathtt{Sec.secureMeth}(\mathtt{shadow\ String}(\odot, \delta))))$

$\mathrm{Shadow}(10, \mathtt{TopLevel.main}(\mathtt{shadow\ String}(\odot, \delta)))$

$\mathrm{Shadow}(11, \mathtt{shadow\ String}(\odot, \delta))$

$\mathcal{R}(\mathtt{p}(\mathtt{new\ String}(''\mathtt{hello\ }'')), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{sanitize}(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ }'').\mathtt{concat}(\mathtt{new\ String}(\circ, ''\mathtt{world}'')))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{sanitize}($
$\mathtt{String.concat}(\mathtt{new\ String}(\bullet, @(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ }'').\mathtt{val}, \mathtt{new\ String}(\circ, ''\mathtt{world}'').\mathtt{val})))))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{sanitize}($
$\mathtt{String.concat}(\mathtt{new\ String}(\bullet, @(''\mathtt{hello\ }'', \mathtt{new\ String}(\circ, ''\mathtt{world}'').\mathtt{val})))))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{sanitize}(\mathtt{String.concat}(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ world}'')))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{sanitize}(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ world}'')))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{Sec.sanitize}(\mathtt{new\ Sec}(\circ).\_\mathtt{sanitize}(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ world}'')).\mathtt{endorse}()))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{Sec.sanitize}(\mathtt{new\ String}(\bullet, ''\mathtt{hello\ world}'').\mathtt{endorse}()))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{Sec.sanitize}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{new\ Sec}(\circ).\mathtt{secureMeth}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{Sec.secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{log}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''));$
$\mathtt{new\ Sec}(\circ).\mathtt{check}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''));\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''))), \varnothing$

$\to \mathtt{TopLevel.main}(\mathtt{Sec.secureMeth}(\mathtt{new\ Sec}(\circ).\mathtt{check}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''));\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''))),$
$\{\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')\}$

$\to \mathtt{TopLevel.main}(\mathtt{Sec.secureMeth}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''))), \{\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')\}$

$\to \mathtt{TopLevel.main}(\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')), \{\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')\}$

$\to \mathtt{new\ String}(\odot, ''\mathtt{hello\ world}''), \{\mathtt{new\ String}(\odot, ''\mathtt{hello\ world}'')\}$

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