A Formal Approach to Combining Prospective and Retrospective Security

A Dissertation Presented

by

Sepehr Amir-Mohammadian

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
Specializing in Computer Science

October, 2017

Defense Date: July 14th, 2017
Dissertation Examination Committee:

Christian Skalka, Ph.D., Advisor
Christelle Vincent, Ph.D., Chairperson
Jeffrey Dinitz, Ph.D.
Byung Lee, Ph.D.
Cynthia J. Forehand, Ph.D., Dean of Graduate College

Abstract

The major goal of this dissertation is to enhance software security by provably correct enforcement of in-depth policies. In-depth security policies allude to heterogeneous specification of security strategies that are required to be followed *before* and *after* sensitive operations. Prospective security is the enforcement of security, or detection of security violations before the execution of sensitive operations, e.g., in authorization, authentication and information flow. Retrospective security refers to security checks after the execution of sensitive operations, which is accomplished through accountability and deterrence. Retrospective security frameworks are built upon *auditing* in order to provide sufficient evidence to hold users accountable for their actions and potentially support other remediation actions. Correctness and efficiency of audit logs play significant roles in reaching the accountability goals that are required by retrospective, and consequently, in-depth security policies. This dissertation addresses correct audit logging in a formal framework.

Leveraging retrospective controls beside the existing prospective measures enhances security in numerous applications. This dissertation focuses on two major application spaces for in-depth enforcement. The first is to enhance prospective security through surveillance and accountability. For example, authorization mechanisms could be improved by guaranteed retrospective checks in environments where there is a high cost of access denial, e.g., healthcare systems. The second application space is the amelioration of potentially flawed prospective measures through retrospective checks. For instance, erroneous implementations of input sanitization methods expose vulnerabilities in taint analysis tools that enforce direct flow of data integrity policies. In this regard, we propose an in-depth enforcement framework to mitigate such problems. We also propose a general semantic notion of explicit flow of information integrity in a high-level language with sanitization.

This dissertation studies the ways by which prospective and retrospective security could be enforced uniformly in a provably correct manner to handle security challenges in legacy systems. Provable correctness of our results relies on the formal Programming Languages-based approach that we have taken in order to provide software security assurance. Moreover, this dissertation includes the implementation of such in-depth enforcement mechanisms for a medical records web application.

To my parents, Nasrin and Sirous,
and the joy of my life, Afsoon

Acknowledgements

My advisor, Prof. Christian Skalka, deserves my sincerest acknowledgement for his dedication, insight and resourcefulness throughout the writing of this dissertation. He helped me develop my skills as a researcher by actively exploring different solutions to the problems and reaching out to other scholars in order to bring in new ideas. I consider myself very fortunate that I have had the opportunity to work with him within the last few years.

I would like to thank Prof. Stephen Chong at the Harvard University, who along with Prof. Skalka came up with the seeds for this dissertation, and collaborated in different aspects of this research. I would also like to thank Prof. Trent Jaeger at the Pennsylvania State University, whose consultation flourished in applying our proposed techniques on real-world problems.

Besides my advisor, my gratitudes are also due to other members of the dissertation defense committee. Thank you to Prof. Christelle Vincent for kindly accepting to be the chair of this committee. Prof. Byung Lee has been a great source of support for me during my doctoral studies, for which I am extremely grateful. Thank you to Prof. Jeffrey Dinitz for agreeing to be a member of this committee not very long before the defense.

I would also like to acknowledge Emeritus Prof. Robert Snapp and Prof. Alan Ling as members of my doctoral studies committee, especially Prof. Snapp who has been an invaluable mentor and has continuously supported me during my studies at the University of Vermont. I would also like to express my gratitude to the other faculty at the Department of Computer Science, including Prof. Joshua Bongard and Prof. Maggie Eppstein.

I consider myself indebted to Prof. Mehran S. Fallah at the Amirkabir University of Technology, who introduced me to the formal aspects of Computer Science during my undergraduate studies, and in a later stage, helped me with a great enthusiasm to take my first steps in doing research in the realm of Programming Languages-based security.

On a rather personal note, I would like to thank my family for their absolute support and companionship: my mother, Nasrin, and my father, Sirous, who have always encouraged me to be ambitious and follow my dreams; my sister, Sepideh, who is a great source of delight in my life; and the love of my life, Afsoon, who has filled my life with joy and happiness.

This dissertation has been supported in part by the United States National Science Foundation, under Grant No. 1408801.

Table of Contents

Dedication ii
Acknowledgements iii
List of Figures ix

1 Introduction **1**
1.1 Foundations of Audit Logging 5
1.2 Application 1: Temporal Properties of Function Calls and Break the Glass Policies 6
1.2.1 A Motivating Example from Practice 8
1.2.2 Threat Model 10
1.3 Application 2: Direct Information Flow and Dynamic Integrity Taint Analysis 10
1.3.1 Vulnerability and Countermeasures 14
1.3.2 Semantics of Dynamic Integrity Taint Analysis 16
1.3.3 The Security and Threat Model 18
1.4 Related Work 20
1.5 Overview and Main Technical Results of this Dissertation 27

2 A Semantics of Audit Logging **31**
2.1 Introduction to Information Algebra 31
2.1.1 Illustrative Example: Relational Algebras 33
2.2 General Model for Logging Specifications 35
2.3 Correctness Conditions for Audit Logs 36
2.4 Correct Logging Instrumentation is a Safety Property 38
2.5 Implementing Logging Specifications with Program Rewriting 39
2.6 Languages for Logging Specifications 42
2.6.1 First Order Logic (FOL) 42
2.6.2 Relational Database 51
2.6.3 Transforming and Combining Audit Logs 56

3 Temporal properties of function calls: Break the glass policies **59**
3.1 Source Language 59
3.1.1 Syntax 60
3.1.2 Semantics 61
3.2 Rewriting Programs with Logging Specifications 64
3.2.1 Specifications Based on Function Call Properties 65
3.2.2 Edit Automata Enforcement of **Calls** Specifications 67
3.2.3 Target Language 68

3.2.4 Program Rewriting Algorithm 70
3.3 Case Study on a Medical Records System 75
3.3.1 Break the Glass Policies for OpenMRS 78
3.3.2 Code Instrumentation 79
3.3.3 Proof Engine 81
3.3.4 Writing and Storing the Log 82
3.4 Reducing Memory Overhead 83
3.4.1 Language with Memory Overhead Mitigation 87
3.4.2 Correctness of Memory Overhead Mitigation 90
3.4.3 An Illustrative Example for Memory Overhead Mitigation . . 99

4 Direct Information Flow: Dynamic Integrity Taint Analysis 103
4.1 An OO Model for Integrity Taint Analysis 105
4.2 In-Depth Integrity Analysis Specified Logically 108
4.2.1 Taint Tracking as a Logical Trace Property 109
4.2.2 Shadow Traces, Taint Propagation, and Sanitization 111
4.2.3 In-Depth Integrity Taint Analysis Policies 114
4.3 Taint Analysis Instrumentation via Program Rewriting 116
4.3.1 In-Depth Taint Analysis Instrumentation 116
4.3.2 Taint Propagation of Library Methods 117
4.3.3 Operational Semantics of $\mathrm{FJ}_{\mathrm{taint}}$ 118
4.3.4 An Illustrative Example: Dynamic Integrity Taint Analysis . . 119
4.4 Properties of Program Rewriting 120
4.4.1 Semantics Preservation 121
4.4.2 Operational Correctness for Prospective Analysis 123
4.4.3 Soundness/Completeness for Retrospective Analysis 124
4.4.4 Definition of $:\approx$ and Correctness Results 125

5 The Meaning of Dynamic Integrity Taint Analysis 140
5.1 Direct Integrity Flow Semantics: Explicit Integrity 141
5.1.1 Model Specification 142
5.2 An OO Model 145
5.2.1 Operational Semantics 146
5.2.2 Model instantiation for FJ 147
5.2.3 Sanity Conditions on Library Methods 149
5.2.4 Illustrative Examples: Incompatibility of Noninterference and Explicit Integrity 151
5.3 Taint Analysis Instrumentation via Program Rewriting 153
5.4 Enforcement of Explicit Integrity Modulo Endorsement by $\mathcal{R}$ 155
5.4.1 Discussion 159

6 Conclusion and Future Work **162**
6.1 Future Work 164

List of Figures

2.1 Properties of Natural Join and Projection 34
2.2 Concept Diagram: Logging Specification and Correctness of Audit Logs. 37

3.1 FJ Syntax 61
3.2 Class Fields in FJ 62
3.3 Method Bodies in FJ 62
3.4 Method Inheritance in FJ 63
3.5 Method Types in FJ 63
3.6 FJ Operational Semantics 64
3.7 Definition of $toFOL(\cdot)$ for Configurations. 64
3.8 Horn Clauses in **Calls** 66
3.9 Edit Automata to Enforce Ideal Instrumentation 68
3.10 $\mathrm{FJ}_{\mathrm{log}}$ Syntax and Sematics Extensions. 69
3.11 Axioms for Rewriting Algorithm $\mathcal{R}_{\mathrm{FJ}}$ 71
3.12 Module Builder 76
3.13 System Architecture 77
3.14 A Simple Break the Glass Policy Specified in the Proof Engine Database. 79
3.15 Specifying Joint Points for Advices. 80
3.16 Querying the Log with SQL. 83
3.17 Precondition rules for $\mathrm{FJ}'_{\mathrm{log}}$. 102
3.18 Logging Specification for Example 3.4.1 102

4.1 Extending FJ for Dynamic Integrity Taint Analysis. 107
4.2 Shadow Values, Expressions, and Evaluation Contexts. 111
4.3 Definition of Mapping *srewrite*. 112
4.4 match Predicate Definition. 113
4.5 Shadow Predicate Definition. 114
4.6 Predicates for Specifying Prospective and Retrospective Properties . 115
4.7 Axioms for Rewriting Algorithm 118
4.8 Operational Semantics of $\mathrm{FJ}_{\mathrm{taint}}$. 119
4.9 Example 4.3.1: Source Trace. 120
4.10 Example 4.3.1: Shadow Expressions. 121
4.11 Example 4.3.1: Target Trace. 122
4.12 Definition of *overlay*. 126
4.13 Definition of *trim*. 127

5.1 Stack-based Operational Semantics for FJ. 147
5.2 Definitions of *control*, *state* and $\langle\cdot,\cdot\rangle$ in FJ. 147

5.3 Definition trust equivalent states in FJ. 148
5.4 Definition of $=_\circ$ Relation. 150
5.5 Example 5.2.1: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States. 152
5.6 Example 5.2.2: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States. 153
5.7 Example 5.2.3: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States. 154
5.8 Axioms for Rewriting Algorithm Restricted to Prospective Measures 155
5.9 Undertainting Example. 160

Chapter 1

Introduction

Academic Programming Languages-based security researchers have mostly considered security controls intended to avoid violations before the execution of sensitive operations. This type of enforcement is called *prospective security* throughout this dissertation. Well-established instances include authorization, authentication, and information flow control tools.

In an authorization system, for example, sensitive operations are defined as accesses to the system resources. A reference monitor mediates such operations for any resource that is requested by a user. Therefore, user's eligibility is studied by the reference monitor before that user gains access to a resource. The reference monitor implements a prospective measure in the enforcement of authorization policy as the eligibility check precedes the access [1].

Information flow control is another major example of prospective security. Information flow analyzers aim to prevent the flow of data in specific circumstances. For instance, for the sake of confidentiality such tools prevent the assignment of "public" data to "secret" data containers. To this end, a flow analyzer needs to mediate every assignment and check the confidentiality level of both sides of an assignment in order to either permit or prevent such action, and thus implements a prospective measure.

Retrospective security is the enforcement of security, or detection of security violations, after program termination or at least after the completion of certain operations in non-terminating processes, e.g., operating systems and server-side programs. [2, 3, 4]. Many real-world systems use retrospective security. For example, the financial industry corrects errors and fraudulent transactions not by proactively preventing suspicious transactions, but by retrospectively correcting or undoing these problematic transactions. Another example is a hospital whose employees are trusted to access confidential patient records, but who might (rarely) violate this trust [5]. Upon detection of such violations, security is enforced retrospectively by holding responsible employees accountable [6].

Retrospective security cannot be achieved entirely by traditional computer security mechanisms, such as access control, or information-flow control. Reasons include that detection of violations may be external to the computer system (such as consumer reports of fraudulent transactions, or confidential patient information appearing in news media), the high cost of access denial (e.g., preventing emergency-room physicians from accessing medical records) coupled with high trust of systems users (e.g., users are trusted employees that rarely violate this trust) [7]. In addition, remediation actions to address violations may also be external to the computer system, such as reprimanding employees, prosecuting law suits, or otherwise holding users accountable for their actions [6].

Auditing underlies retrospective security frameworks and has become increasingly important to the theory and practice of cybersecurity. By recording appropriate aspects of a computer system's execution an audit log (and subsequent examination of the audit log) can enable detection of violations, and provide sufficient evidence to hold users accountable for their actions and support other remediation actions. For example, an audit log can be used to determine *post facto* which users performed dangerous operations, and can provide evidence for use in litigation.

However, despite the importance of auditing to real-world security, relatively little work has focused on the formal foundations of auditing, particularly with respect to defining and ensuring the correctness of audit log generation. Indeed, correct and efficient audit log generation poses at least two significant challenges. First, it is necessary to record sufficient and correct information in the audit log. If a program is manually instrumented, it is possible for developers to fail to record relevant events. Recent work showed that major health informatics systems do not log sufficient information to determine compliance with HIPAA policies [8]. Second, an audit log should ideally not contain more information than needed. While it is straightforward to collect sufficient information by recording essentially *all* events in a computer system, this can cause performance issues, both slowing down the system due to generating massive audit logs, and requiring the handling of extremely large audit logs. Excessive data collection is a key challenge for auditing [9, 10, 11], and is a critical factor in the design of tools that generate and employ audit logs (e.g., spam filters [12]).

Retrospective security is usually employed as an additional controlling mechanism to enhance the effectiveness of prospective enforcement, rather than being used in isolation. For example, audit logging is used in authorization systems to study the details of access to resources in a post facto manner. This helps identifying potential vulnerabilities in the design and implementation of access control, the authorization policy, etc. In this regard, combining prospective and retrospective security needs to be investigated in its effectiveness, i.e., the guarantees provided by the combination of these two approaches is required to be studied. To this end, we formally specify the correctness notions for prospective and retrospective measures and provide a framework to enforce both in a unified fashion such that the enforcement is correct. This facilitates establishing formal guarantees for combined prospective and retrospective enforcements and paves the way for correct implementations.

To avoid confusion in our formal presentation and discussion we define some terminology. These definitions are also intended to better isolate and describe elements of auditing, in particular we note that auditing of processes typically involves two distinct policies: one for generating the audit log itself, and one for analyzing the audit log. An *execution trace* is a complete record of program execution in a formal operational semantics. An *audit log* is a record of program execution. The format of audit logs varies, but essentially comprises the information derived from the full execution trace. A *log query* is any question that can be asked of the audit log, though it is typically said that the program execution is "being audited" in this case. This reveals an implicit understanding that audit logs bear a knowable relation with processes. A *logging specification* specifies how audit logs should be generated, that is, what would be the relation between the information contained in the execution trace and in the audit log. Logging specifications should typically ensure that audit logs contain enough information, and in an appropriate form, to answer a given log query (or set of log queries). A logging specification is *enforced* by a program if execution of the program produces an appropriate audit log. In this dissertation, we are interested in formally defining logging specifications, and automatically enforcing them.

A *rewriting algorithm* takes a logging specification and a program, and instruments the program to enforce the logging specification. A rewriting algorithm may support only a limited class of logging specifications. An *auditing policy* is the combination of a log query and a logging specification. Thus, an auditing policy describes both a logging specification to enforce, and a query to ask of the resulting audit logs. We refer to the combination of prospective and retrospective measures in the enforcement of security policies as *in-depth (or, heterogeneous) policy enforcement.*

In the following, we introduce different aspects of this dissertation. The reader is referred to Section 1.5 for a summary of the technical results.

1.1 Foundations of Audit Logging

A main goal of this dissertation is to establish a formal foundation for audit logging, especially to establish general correctness conditions for audit logs and logging instrumentation. We define a general semantics of audit logs using the theory of *information algebra* [13]. We interpret both program execution traces and audit logs as information elements in an information algebra. A *logging specification* defines the intended relation between the information in traces and in audit logs. An audit log is correct if it satisfies this relation. A benefit of this formulation is that it separates logging specifications from programs, rather than burying them in code and implementation details.

Separating logging specifications from programs supports clearer definitions and more direct reasoning. Additionally, it enables algorithms for implementing general classes of logging specifications. Our formal theory establishes conditions that guarantee enforcement of logging specifications by such algorithms. As we will show, correct instrumentation of logging specifications is a safety property, hence enforceable by security automata [1]. Inspired by related approaches to security automata implementation [14], we focus on program rewriting to automatically enforce correct audit instrumentation. Program rewriting has a number of practical benefits versus, for example, program monitors, such as lower OS process management overhead.

This approach would allow system administrators to define logging specifications which are automatically instrumented in code, including legacy code. Implementation details and matters such as optimization can be handled by the general program rewriting algorithm, not the logging specification. Furthermore, establishing correctness of a program rewriting algorithm provides an important security guarantee. Such an algorithm ensures that logging specifications will be implemented correctly, even if the rewritten source code contains malicious code or programmer errors. The reader may refer to Chapter 2 for the details of logging semantics. In what follows, we introduce two major applications for our framework.

1.2 Application 1: Temporal Properties of Function Calls and Break the Glass Policies

As mentioned in Section 1.1, separating logging specifications from program code enables support for classes of logging specifications. A general class of logging specifications that supports different auditing policies is the one that captures *temporal* properties of function invocations. Function call as an event is a key factor in logging and is considered as the fundamental unit of "secure operations" in many systems, e.g., in OpenMRS [15], which is a popular open source medical records software system. Break the glass policies, for instance, can be specified in terms of temporal relation between function calls and therefore this particular class of logging specifications includes this set of policies.

Break the glass policies enable users to temporarily boost their authority in the system and thus access to system resources, for which they do not possess the required access rights according to the authorization policy. However, break the glass policies associate such unrestricted access with retrospective checks, i.e., by breaking the glass, the user asserts her consent to be audited in a later stage. Therefore, breaking the glass entails audit log generation for the given user from that point forward. This per se represents a temporal relation between system events, in particular the action of breaking the glass by some user and the subsequent recording of what user does. Indeed, these policies can be extended with additional preconditions, e.g., certain users may only gain a limited extent of authority. Break the glass policies are common in critical systems, e.g., information systems that are employed for medical and firefighting purposes extensively use these policies in critical cases. For more information on break the glass policies the reader may refer to [16]. Section 1.2.1 gives an example break the glass policy.

We provide a program instrumentation algorithm that rewrites programs to support logging specifications that pertain to temporal properties of function invocations. We consider a case study of our approach, a program rewriting algorithm for correct instrumentation of logging specifications in OpenMRS. Our tool allows system administrators to define logging specifications which are automatically instrumented in OpenMRS legacy code. Implementation details and optimizations are handled transparently by the general program rewriting algorithm, not the logging specification. Formal foundations ensure that logging specifications are implemented correctly by the algorithm. In particular, we show how our system can implement break the glass auditing policies. The details of formalization and deployment for logging specifications that address temporal properties of function calls is given in Chapter 3.

In the following we informally discuss a motivating example that introduces break the glass policies, and identify our threat model.

1.2.1 A Motivating Example from Practice

Although audit logs contain information *about* program execution, they are not just a straightforward selection of program events. Illustrative examples from practice include break the glass policies used in electronic medical record systems [16]. These policies use access control to disallow care providers from performing sensitive operations such as viewing patient records, however care providers can "break the glass" in an emergency situation to temporarily raise their authority and access patient records, *with the understanding that subsequent sensitive operations will be logged and potentially audited.* One potential accountability goal is the following:

> *In the event that a patient's sensitive information is inappropriately leaked, determine who accessed a given patient's files due to "breaking the glass."*

Since it cannot be predicted a priori whose information may leak, this goal can be supported by using an audit log that records all reads of sensitive files following glass breaking. To generate correct audit logs, programs must be instrumented for logging appropriately, i.e., to implement the following *logging specification* that we call LS_H:

> LS_H : *Record in the log all patient information file reads following a break the glass event, along with the identity of the user that broke the glass.*

If at some point in time in the future it is determined that a specific patient **P**'s information was leaked, logs thus generated can be analyzed with the following query that we call LQ_H:

> LQ_H : *Retrieve the identity of all users that read **P**'s information files.*

The specification LS_H and the query LQ_H together constitute an auditing policy that directly supports the above-stated accountability goal. Their separation is useful since at the time of execution the information leak is unknown, hence **P** is not known. Thus while it is possible to implement LS_H as part of program execution, LQ_H must be implemented retrospectively.

It is crucial to the enforcement of the above accountability goal that LS_H is implemented correctly. If logging is incomplete then some potential recipients may be missed. If logging is overzealous then bloat is possible and audit logs become "write only". These types of errors are common in practice [8]. To establish formal correctness of instrumentation for audit logs, it is necessary to define a formal language of logging specifications, and establish techniques to guarantee that instrumented programs satisfy logging specifications. That is one of the concerns of this dissertation. Other work has focused on formalisms for querying logs [17, 18], however these works presuppose correctness of audit logs for true accountability.

Another practical goal of our work is to allow logging specifications that minimize logging overhead. In the case of LS_H, this requires that program trace characteristics be expressible. Later (see Example 2.6.1) we revisit this and show how LS_H can be formally specified and enforced in a proposed system. However, given the diversity of auditing practice, another goal is generality, so that our foundations have breadth and support reasoning about the interoperations of different systems. We return to this discussion in Section 2.6.

1.2.2 Threat Model

With respect to program rewriting (i.e., automatic techniques to instrument existing programs to satisfy a logging specification), we regard the program undergoing instrumentation as untrusted. That is, the program source code may have been written to avoid, confuse, or subvert the automatic instrumentation techniques. We do, however, assume that the source code is well-formed (valid syntax, well-typed, etc.). Moreover, we trust the compiler, the program rewriting algorithm, and the runtime environment in which the instrumented program will ultimately be executed. Non-malleability of generated audit logs, while important, is beyond the scope of this work.

1.3 Application 2: Direct Information Flow and Dynamic Integrity Taint Analysis

Dynamic taint analysis implements a “direct” or “explicit” information flow analysis to support a variety of security mechanisms. Similar to information flow, taint analysis can be used to support either confidentiality or integrity properties. An important application of integrity taint analysis is to prevent the execution of security sensitive operations on untrusted data, in particular to combat cross-site scripting (XSS) and SQL injection attacks in web applications [19]. 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 [19, 20]. For instance in a real-world news managements system, 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 dissertation. Algorithm 1 refers to such imperfect sanitization in a real world news manage system, written in PHP [20]. In this code excerpt, the low integrity input is stored in the variable `userid`. This data needs to be sanitized before being used as part of SQL query in line 16. The sanitization must ensure that `userid` refers to a string of digits. The programmer aims to do so in line 12, by checking whether the string identified by `userid` is in the regular language defined by regex `[0-9]+`. However, `[0-9]+` refers to the regular language of strings in which *at least one digit* exists. The programmer has mistakenly missed to use $ and ˆ in the regex definition to limit the language corresponding to that regex to only contain strings of digits. This bug results in SQL command injection vulnerability.

Algorithm 1: A Real World Example of Imperfect Sanitization [20].

```
isset ($_GET['userid']) ?
      $userid = $_GET['userid'] : $userid = '';
if ($USER['groupid'] != 1) then
    // permission denied
    unp_msg($gp_permserror);
    exit;
end
if ($userid == '') then
    unp_msg($gp_invalidrequest);
    exit;
end
if (!eregi('[0-9]+',$userid)) then
    unp_msg('You entered an invalid user ID.');
    exit;
end
$getuser = $DB->query("SELECT * FROM 'unp_user' WHERE
userid='$userid"');
if (!$DB->is_single_row($getuser)) then
    unp_msg('You entered an invalid user ID.');
    exit;
end
```

Thus, a main challenge we address is how to mitigate imperfect sanitization in taint analysis. Our solution is an *in-depth* approach [21]– 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 [22, 23], an existing Java taint analysis system with empirically demonstrated efficiency, and our approach to retrospective security is aimed to minimize the size of logs. The theoretical foundations we establish in this dissertation 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 (introduced in Section 1.2). However, the formulations proposed here could be applied more broadly.

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. We also establish 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 "operationally correct" with respect to the prospective policy specification and sound and complete with respect to the retrospective policy enforcement. This algorithm and the proofs of operational correctness for prospective enforcement and soundness/completeness of retrospective enforcement are formulated with respect to an idealized language model.

Beside operational correctness, we propose a semantic framework to model direct flow of data integrity enforced by integrity taint analysis techniques, called *explicit integrity*. Explicit integrity defines the security property that dynamic integrity taint analysis tools need to enforce. In this regard, explicit integrity is related to (but incompatible) noninterference. This framework is based on explicit secrecy [24], which studies direct flow of information for confidentiality purposes. We use this semantics to establish provable correctness conditions for rewriting algorithms that instrument integrity taint analysis in the presence of input sanitization. This way an underlying semantic framework is provided to study numerous other integrity taint analyzers in the future. We prove that our rewriting algorithm is correct using explicit integrity. Explicit integrity demonstrates an underlying model for direct flow of information in functional calculi, which to the best of our knowledge has not been studied before.

1.3.1 Vulnerability and Countermeasures

Our work is significantly inspired by a previously unreported security flaw in OpenMRS. This flaw allows an attacker to launch persistent XSS attacks[1]. When a web-based software receives and 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 (e.g., `PersonNameValidator`, `VisitTypeValidator`, etc.). For some of these classes the implementation is strict enough to reject script tags by enforcing data to match a particular regular expression, e.g., `PersonNameValidator`. However, `VisitTypeValidator` lacks such restriction and only checks for object fields to avoid being null, empty or whitespace, and their lengths to be correct. Thus the corresponding webpage that receives user inputs to construct `VisitType` objects (named `VisitTypeForm.jsp`) is generally not able to perform proper sanitization through the

[1]We have responsibly disclosed the vulnerabilities we have found in OpenMRS (version 2.4, released 7/15/2016, and the preceding versions) to the OpenMRS developing community. We discuss one particular case here.

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 [19, 20].

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 [22, 23] 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 [25, 26]). Operational correctness of the algorithm is proven in an idealized model of Phosphor-style taint analysis defined for a "Featherweight" Java (FJ) core language [27]. 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. Operation correctness is established in terms of the compliance of the instrumentation to the policy specification.

1.3.2 Semantics of Dynamic Integrity Taint Analysis

Taint analysis has many applications [19, 28, 29], but its security semantics is not well understood. Taint analysis is typically defined as a direct, aka explicit, information flow variant, where indirect flow of information is ignored. The semantics of information flow [30] has been well studied and is typically characterized via noninterference properties [31], but surprisingly little work has been done to develop similar properties for taint analysis. Formal definitions of taint analysis implementations do exist, but they are usually operational in nature [32, 33].

An exception to this rule is work on explicit secrecy [24], which explores the semantics of dynamic confidentiality taint analysis. However, this previous work only considers taint analysis for low level languages with a greater focus on data *confidentiality.* Influenced by this model, we propose the semantics for dynamic *integrity* taint analysis, which is defense against injection attacks in high level code such as web applications [19, 34]. The issue here is ensuring that low integrity data sources do not directly affect high integrity sinks such as security sensitive operations.

As introduced in Section 1.3.1, we have considered formal specifications of taint analysis for hardening Java programs against injection attacks, with a particular interest in hardening a web-based medical records software system. The formulation studied here provides a meaningful semantics for the dynamic taint analysis that we have defined, which is a "core" formulation of Phosphor taint analysis. However, in addition to Phosphor, our approach provides a semantic foundation for reasoning about other existing integrity dynamic taint analysis in high level languages (HLLs) [35], as well as low level languages.

The main technical contribution in this realm is a definition of the security property enforced by dynamic integrity taint analysis. This property, called *explicit integrity*, is general in that it applies to both high and low level languages. Our model is based on previous work on explicit secrecy [24] for low level languages. We demonstrate how that model can be applied for data integrity purposes in HLLs. Information integrity can often be expressed as the dual of confidentiality [36]. However, the dual of explicit secrecy does not support either the integrity flows in HLLs with structured data, or the functional paradigm. Explicit integrity comports with applications of dynamic taint analysis to harden programs against injection attacks– where low integrity user data may maliciously taint arguments to security sensitive operations, e.g., database interactions in the case of SQL injection.

In addition to the basic definition, we also extend our notion of explicit integrity to support *endorsement*, which allows dynamic "untainting" of data usually as a result of sanitization [20, 37]. Endorsement is usually considered the integrity dual of confidentiality declassification, and we are able to adapt techniques for the latter formulated for explicit secrecy.

Aside from providing general semantic insights and formal foundations for dynamic integrity taint analyses, we also show how explicit integrity provides guiding principles for efficiency strategies in taint analysis instrumentation. In particular, system designers often treat library methods as black boxes with respect to taint propagation, i.e. taint properties are directly ascribed to library method results [25, 26]. For example, some systems will not ascribe taint to the result of character projection from tainted strings [26], and other systems will ascribe taint to the results if arguments are tainted [25], regardless of direct flow internal to the methods. The former is an example of "undertainting", while the latter is an example of "overtainting". Our formal account provides a precise characterization of these informal notions (see Definitions 5.2.2 and 5.2.3), and a basis for designers to understand the effects of implementation decisions with respect to a higher level security property.

1.3.3 The Security and Threat 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 over primitive objects, so there is no question about which argument may be tainted. Since we define a Java based model, each sso or sanitizer is identified as a specific method `m` in a class `C`. That is, there exists a set of *Sanitizers* containing class, method pairs `C.m` which are assumed to return high-integrity data, though they may be passed low-integrity data. Likewise, there exists a set of *SSOs* of the same form, and for brevity we will write *sso*(`e`) for a method invocation `new v.m(e)` on some object `v` where `C.m` $\in$ *SSO*. Note that *SSOs* and *Sanitizers* are assumed to be closed under class inheritance, where the method is not overridden. As a sanity condition we require $\mathit{SSOs} \cap \mathit{Sanitizers} = \varnothing$. For simplicity of our formal presentation we assume that only one tainted source will exist. Explicit integrity, as a high-level property, is instantiated for this model.

We assume that our program rewriting algorithm is trusted. Input code is trusted to be not malicious, though it may contain errors. We note that this assumption is important for application of taint analysis that disregards indirect flows, since there is confidence that the latter will not be 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.

1.4 Related Work

Previous work by DeYoung et al. has studied audit policy specification for medical (HIPAA) and business (GLBA) processes [38, 39]. This work illustrates the effectiveness and generality of a temporal logic foundation for audit policy specification, which is well-founded in a general theory of privacy [18]. Their auditing system has also been implemented in a tool similar to an interactive theorem prover [40]. Their specification language inspired our approach to logging specification semantics. However, this previous work assumes that audit logs are given, and does not consider the correctness of logs. Some work does consider trustworthiness of logs [41], but only in terms of tampering (malleability). In contrast, our work provides formal foundations for the correctness of audit logs, and considers algorithms to automatically instrument programs to generate correct logs.

Other work applies formal methods (including predicate logics [42, 43], process calculi and game theory [44]) to model, specify, and enforce auditing and accountability requirements in distributed systems. In that work, audit logs serve as evidence of resource access rights, an idea also explored in Aura [17] and the APPLE system [45]. In Aura, audit logs record machine-checkable proofs of compliance in the Aura policy language. APPLE proposes a framework based on trust management and audit logic with log generation functionality for a limited set of operations, in order to check user compliance.

In contrast, we provide a formal foundation to support a broad class of logging specifications and relevant correctness conditions. In this respect our proposed system is closely related to PQL [46], which supports program rewriting with instrumentation to answer queries about program execution. From a technical perspective, our approach is also related to trace matching in AspectJ [47], especially in the use of logic to specify trace patterns. However, the concern in that work is aspect pointcut specification, not logging correctness, and their method call patterns are restricted to be regular expressions with no conditions on arguments, whereas the latter is needed for the specifications in temporal properties of function calls.

Logging specifications are related to safety properties [1] and are enforceable by security automata, as we have shown. Hence IRM rewriting techniques could be used to implement them [14]. However, the theory of safety properties does not address correctness of audit logs as we do, and our approach can be viewed as a logging-specific IRM strategy. Guts et al. [48] develop a static technique to guarantee that programs are properly instrumented to generate audit logs with sufficient evidence for auditing purposes. As in our research, this is accomplished by first defining a formal semantics of auditing. However, they are interested in evidence-based auditing for specific distributed protocols.

Other recent work [9] has proposed log filters as a required improvement to the current logging practices in the industry due to costly resource consumption and the loss of necessary log information among the collected redundant data. This work is purely empirical, not foundational, but provides practical evidence of the relevance of our efforts since logging filters could be defined as logging specifications.

Audit logs can be considered a form of *provenance*: the history of computation and data. Several recent works have considered formal semantics of provenance [49, 50]. Cheney [51] presents a framework for provenance, built on a notion of system traces. Recently, W3C has proposed a data model for provenance, called PROV [52], which enjoys a formal description of its specified constraints and inferences in first-order logic, [53], however the given semantics does not cover the relationship between the provenance record and the actual system behavior. The confidentiality and integrity of provenance information is also a significant concern [54].

Taint analysis is an established solution to enforce confidentiality and integrity policies through direct data flow control. Various systems have been proposed for both low and high level level languages. Our policy language and semantics are based on a well-developed formal foundation, where we interpret Horn clause logic as an instance of information algebra [55] in order to specify and interpret retrospective policies.

Schwartz et al. [32] define a general model for runtime enforcement of policies using taint tracking for an intermediate language. In Livshits [33], taint analysis is expressed as part of operational semantics, similar to Schwartz et al. [32], and a taxonomy of taint tracking is defined. Livshits et al. [19] 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 [56] 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 [19, 20], incomplete input sanitization makes a variety of applications susceptible to injection attacks.

Another related line of work is focused on the optimization of integrity taint tracking deployment in web-based applications. Sekar [34] 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. [57] attempt to lower the memory overhead of TaintDroid taint tracker [28] 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. [58] 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. [59] 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. [60] 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. [59], they use shadow memory to keep taint information, with a fixed offset from user memory space. Zhu et al. [29] 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. [26] 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. [25] propose an object-level tainting mechanism for Java strings. They study the same classes as the ones in Chin et al. [26], 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 [22, 23] is an attempt to apply taint tracking more generally in Java, to any primitive type and object class. Phosphor instruments the application and libraries at bytecode level based on a given list of taint source and sink methods. Input sanitizers with endorsement are not directly supported, however. As Phosphor avoids any modifications to the JVM, the instrumented code is still portable. Our work is an attempt to formalize Phosphor in FJ extended with input sanitization and in-depth enforcement. Our larger goal is to develop an implementation of in-depth dynamic integrity analysis for Java by leveraging the existing Phosphor system.

Secure information flow [61] and its interpretation as the well-known hyperproperty [62] of noninterference [31] is challenging to implement in practical settings [30] due to implicit flows. Taint analysis is thus an established solution to enforce confidentiality and integrity policies since it tracks only direct data flow control. Various systems have been proposed for both low and high level level languages. The majority of previous work, however, has been focused on taint analysis policy specification and enforcement (e.g., [32, 33, 57, 63]), rather than capturing the essence of direct information flow which could provide an underlying framework to study numerous taint analysis tools.

Knowledge-based semantics has been introduced by Askarov et al. [64] as a general model for information flow of confidential data, concentrated on cryptographic computations and key release (declassification [65]) and later employed in other data secrecy analyses [37, 66, 67]. Recently, Schoepe et al. [24] have proposed the semantic notion of correctness for taint tracking that enforces confidentiality policies of direct information flow, called explicit secrecy. To this end, they propose a knowledge-based semantics, influenced by Volpano's weak secrecy [68] and gradual release [64]. Explicit secrecy is defined as a property of a program, where the program execution does not change the explicit knowledge of public user. The authors show that noninterference is not comparable to explicit secrecy. However, rather than restricting the discussion to direct information flow in a low level language, we model a high level OO language with a functional flavor to represent generality of our framework.

A counterpart for attacker knowledge in the realm of general flow of information integrity, called attacker power [37], is introduced as the set of low integrity inputs that generate the same observables. In this regard, Askarov et al. [37] use holes in the syntax of program code for injection points, influenced by [69]. However, their attack model is different as the low integrity and low confidentiality user is able to inject program code in the main program, by which she could gain more knowledge. We have tailored attacker power for explicit flows using state transformers, in order to interpret integrity taint analysis.

Birgisson et al. [36] give a unified framework to capture different flavors of integrity, in particular integrity via information flow and via different types of invariance. Similar to other works in this line, they give a simple imperative language with labeled operational semantics in order to enforce integrity policies through communication with a monitor. In contrast, we use program rewriting techniques to enforce policies regarding flow of data integrity, which are applicable to legacy systems.

1.5 Overview and Main Technical Results of this Dissertation

This dissertation comprises solutions to tackle the problems regarding the enforcement of in-depth policies. In particular, we propose the first semantic framework for audit logging which enables us to discuss the soundness and completeness notions of retrospective enforcement of security along with the prospective measures. We instantiate this framework with a sufficiently expressive structure to support two major application spaces: First, we propose a provably correct enforcement technique for a general class of logging specifications that could be used to enhance prospective measures through accountability and surveillance. Second, we propose a unified language to express and enforce in-depth policies that aim to ameliorate potentially erroneous prospective controls. This dissertation also discusses a novel semantic framework for dynamic integrity taint analysis, called explicit integrity, that is general enough to be used in both imperative and functional paradigms. The major contributions of this dissertation are as follows:

- The first semantic notion for retrospective enforcement that enables discussing sound (Definition 2.5.2) and complete (Definition 2.5.3) audit logs.
- A rewriting algorithm to enforce retrospective measures (Definition 3.2.3), and prove its soundness and completeness (Theorem 3.2.3).
- The first logical assertion of taint analysis that supports uniform specification and enforcement of the in-depth dynamic integrity taint analysis policies (Definition 4.2.3).

- The first semantic framework for dynamic integrity taint analysis that is inclusive enough to support functional languages along with imperative ones (Definition 5.1.3).

In Chapter 2 we define a semantics of auditing, and establish conditions for correctness of audit rewriting algorithms. That is, we define what it means for a program instrumentation to correctly log information. In Section 2.1, we introduce information algebra as the basis of our model for correct audit log generation. We characterize logging specifications and correctness conditions for audit logs, in a high-level manner using information algebra. In particular, we obtain formal notions of soundness and completeness of program rewriting for auditing (Definitions 2.5.2 and 2.5.3). In Section 2.6, we study the instantiations of the auditing semantics and transformation of these instantiations to each other. In this section, we formulate information algebras based on first-order logic (FOL) and relational algebra that are shown to satisfy necessary conditions (Theorem 2.6.1 and Corollary 2.6.1) to enjoy information-algebraic properties, including a partial information order. We then leverage FOL-based information algebra to define a formal semantics of program auditing (Definition 2.6.5).

Chapter 3 discusses temporal properties of function calls as a general class of logging specifications and break the glass policies. In Section 3.1, the language model specification is given, which is Featherweight Java. In Section 3.2, we consider this particular class of logging specifications and present a rewriting algorithm to support this class In this section, we also prove that a rewriting algorithm is sound and complete with respect to a specific class of logging specifications (Theorem 3.2.3). This illustrates how our auditing semantics can be leveraged to prove program instrumentation correctness for particular rewriting algorithms. In Section 3.3, we discuss a case study on health informatics, particularly OpenMRS system. In this section, we discuss the deployment of correct audit logging mechanism for OpenMRS system, and propose techniques to reduce memory overhead.

Chapter 4 discusses dynamic integrity taint analysis as another application for in-depth enforcement of security. In Section 4.1, we define the source language model for dynamic integrity taint analysis. In Section 4.2, we introduce a novel 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 4.2.3 which specifies the in-depth policy. In Section 4.3, an implementation of dynamic integrity taint analysis is defined as a program rewriting algorithm. This section also contains an extended example in Section 4.3.4 illustrating the main ideas of our formulations. In Section 4.4 the rewriting algorithm is proven operationally correct on the basis of the prospective policy specification, and sound/complete with respect to the retrospective policy specification. The main results are that rewritten programs are simulations of the source with integrity flow reflected in the operational semantics (Theorem 4.4.1) , and that prospective and retrospective policies are correctly instrumented (Theorems 4.4.2 and 4.4.3).

Chapter 5 focuses on the underlying meaning of dynamic integrity taint analysis. In Section 5.1, we define the semantic framework for dynamic integrity taint analysis. We instantiate the semantic framework for the language model in Section 5.2. We also introduce required properties for user-defined taint propagation policies that could avoid incorrect implementations. Section 5.2.4 contains examples illustrating the main ideas of our formulations, in particular incomparability of noninterference with explicit integrity. Section 5.3 describes an implementation of dynamic integrity taint analysis as a program rewriting algorithm. In Section 5.4, we show that our enforcement mechanism satisfies the semantic property for dynamic integrity taint analysis (Theorem 5.4.1).

Chapter 6 concludes the dissertation.

Chapter 2

A Semantics of Audit Logging

Our goal in this chapter is to formally characterize logging specifications and correctness conditions for audit logs [70]. To obtain a general model, we leverage ideas from the theory of *information algebra* [13, 71], which is an abstract mathematical framework for information systems. In short, we interpret program traces as information, and logging specifications as functions from traces to information. This separates logging specifications from their implementation in code, and defines exactly the information that should be in an audit log. This in turn establishes correctness conditions for audit logging implementations.

2.1 Introduction to Information Algebra

Information algebra is the algebraic study of the theory of information. In information algebra, information is seen as a collection of separate elements. Each information element can be queried for further refinement and also aggregated with other information elements. To this end, the algebra consists of two domains: an information domain and a query domain. The information domain Φ is the set of information elements that can be aggregated in order to build more inclusive information elements. The query domain E is a lattice of querying sublanguages in which the partial order relation among these sublanguages represents the granularity of the queries. In order to aggregate and query the information elements, the following operations are defined.

Definition 2.1.1 *Any information algebra (Φ, E) includes two basic operators:*

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

The two-sorted algebra (Φ, E) is an information algebra if the combination and focusing operations defined in Definition 2.1.1 meet specific properties.

Definition 2.1.2 *Any two-sorted algebra (Φ, E) with operators $\otimes : \Phi \times \Phi \to \Phi$ and $\Rightarrow: \Phi \times E \to \Phi$ is an information algerba iff the following properties hold:*

- *Semigroup: Φ is a commutative semigroup under combination, i.e., associativity and commutativity hold for $\otimes$ and there exists a neutral element $I \in \Phi$,*
- *Transitivity of focusing: $(X^{\Rightarrow L})^{\Rightarrow M} = X^{\Rightarrow L \wedge M}$ for all $X \in \Phi$ and $L, M \in E$*[1]*,*
- *Combination: $(X^{\Rightarrow L} \otimes Y)^{\Rightarrow L} = X^{\Rightarrow L} \otimes Y^{\Rightarrow L}$ for all $X, Y \in \Phi$ and $L \in E$,*

[1]Note that $L \wedge M$ refers to the meet of L and M in the lattice E.

- *Support: For all $X \in \Phi$, there exists some $L \in E$ such that $X^{\Rightarrow L} = X$, and*

- *Idempotence: $X \otimes X^{\Rightarrow L} = X$ for all $X \in \Phi$ and $L \in E$.*

Using the combination operator we can define a partial order relation on Φ to compare the information contained in the elements of Φ. A partial ordering is induced on Φ by the so-called *information ordering* relation $\leq$, 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.

Definition 2.1.3 *X is contained in Y, denoted as $X \leq Y$, for all $X, Y \in \Phi$ iff $X \otimes Y = Y$.*

Definition 2.1.4 *We say that X and Y are* information equivalent*, and write $X = Y$, iff $X \leq Y$ and $Y \leq X$.*

For a more detailed account of information algebra, the reader is referred to a definitive survey paper [71]. In what follows, we give an intuitive example of information algebra, before instantiating other structures that we will employ for retrospective security semantics.

2.1.1 Illustrative Example: Relational Algebras

Relational algebra is a well-recognized instance of information algebra [71]. In the following, we briefly introduce relational algebra and discuss this instantiation.

$$(R \bowtie R') \bowtie R'' = R \bowtie (R' \bowtie R'') \qquad R \bowtie R' = R' \bowtie R \qquad \emptyset \bowtie R = R \bowtie \emptyset = R$$

$$\pi_{\mathcal{A}_1}(\pi_{\mathcal{A}_3}(R)) = \pi_{\mathcal{A}_1 \cap \mathcal{A}_3}(R) \qquad \pi_{\mathcal{A}_1}(\pi_{\mathcal{A}_1}(R) \bowtie R') = \pi_{\mathcal{A}_1}(R) \bowtie \pi_{\mathcal{A}_1}(R')$$

$$\text{if } dom(R) = \mathcal{A}_1 \text{ then } \pi_{\mathcal{A}_1}(R) = R \qquad R \bowtie \pi_{\mathcal{A}_1}(R) = R$$

Figure 2.1: Properties of Natural Join and Projection

Relational Algebra

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

Function $f[\mathcal{A}_2] : \mathcal{A}_2 \to \{x_1, ..., x_n\}$ is the *restriction* of f to $\mathcal{A}_2$, defined as $f[\mathcal{A}_2](a) = f(a)$, for all $a \in \mathcal{A}_2$.

A *relation* R over $\mathcal{A}_1$ is a set of functions f defined on a specific set of attributes $\mathcal{A}_1$.

Then, the *projection* of R on $\mathcal{A}_2$ is defined as $\pi_{\mathcal{A}_2}(R) = \{f[\mathcal{A}_2] \mid f \in R\}$.

The *natural join* of relation R over $\mathcal{A}_1$ and R' over $\mathcal{A}_3$ is defined as $R \bowtie R' = \{f \mid dom(f) = \mathcal{A}_1 \cup \mathcal{A}_3, f[\mathcal{A}_1] \in R, f[\mathcal{A}_3] \in R'\}$.

Instantiation

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

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

since the projection and natural join satisfy the properties specified in Figure 2.1.

2.2 General Model for Logging Specifications

Following [1], an *execution trace* $\tau = \kappa_0\kappa_1\kappa_2\ldots$ is a possibly infinite sequence of configurations κ that describe the state of an executing program. We deliberately leave configurations abstract, but examples abound and we explore a specific instantiation for FJ-based calculus in Section 3.2. 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.

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 $\lfloor\sigma\rfloor \leq \lfloor\tau\rfloor$. We intentionally leave both Φ and $\lfloor\cdot\rfloor$ underspecified for generality, though application of our formalism to a particular logging implementation requires instantiation of them. We discuss an example in Section 2.6.

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

We assume that auditing is implementable, requiring at least that all conditions for logging any piece of information must be met in a finite amount of time. As we will show, this restriction implies that correct logging instrumentation is a safety property [1].

Definition 2.2.1 *We require of any logging specification LS that for all traces τ and information $X \leq LS(\tau)$, there exists a finite prefix σ of τ such that $X \leq LS(\sigma)$.*

It is crucial to observe that some logging specifications may *add* information not contained in traces to the auditing process. Security information not relevant to program execution (such as ACLs), interpretation of event data (statistical or otherwise), etc., may be added by the logging specification. For example, in the OpenMRS system [72], logging of sensitive operations includes a human-understandable "type" designation which is not used by any other code. Thus, given a trace τ and logging specification LS, it is *not* necessarily the case that $LS(\tau) \leq \lfloor \tau \rfloor$. Audit logging is not just a filtering of program events.

2.3 Correctness Conditions for Audit Logs

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

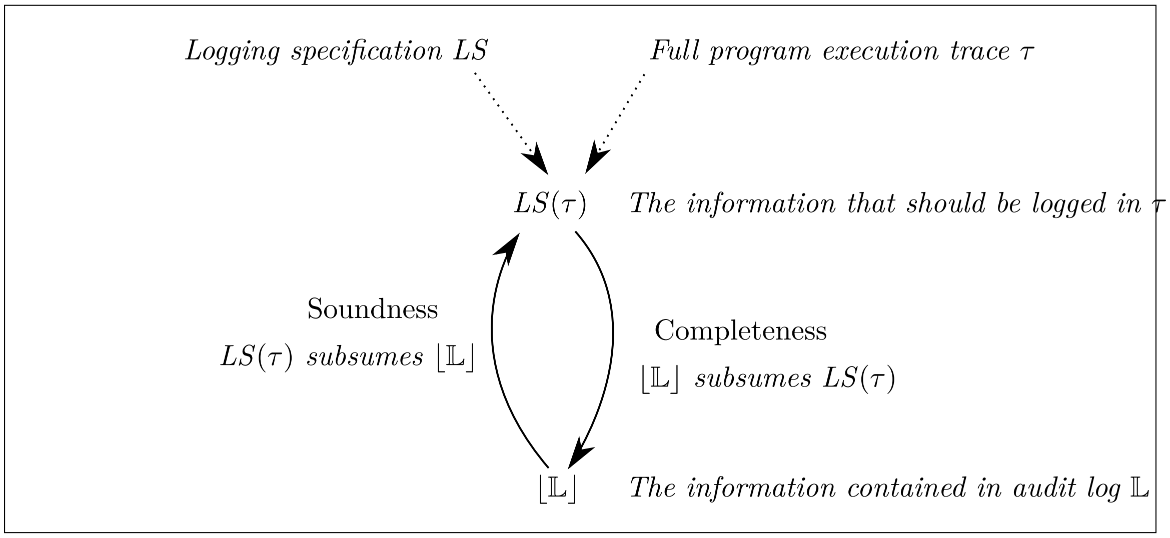
Logging specification LS
Full program execution trace τ
$LS(\tau)$
The information that should be logged in τ
Soundness
$LS(\tau)$ *subsumes* $\lfloor \mathbb{L} \rfloor$
Completeness
$\lfloor \mathbb{L} \rfloor$ *subsumes* $LS(\tau)$
$\lfloor \mathbb{L} \rfloor$
The information contained in audit log $\mathbb{L}$

Figure 2.2: Concept Diagram: Logging Specification and Correctness of Audit Logs.

to Φ. Intuitively, $\lfloor \mathbb{L} \rfloor$ denotes the information in $\mathbb{L}$, interpreted as an element of Φ.

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

Definition 2.3.1 *Audit log* $\mathbb{L}$ *is* sound with respect to logging specification LS and execution trace τ *iff* $\lfloor \mathbb{L} \rfloor \leq LS(\tau)$.

Definition 2.3.2 *Audit log* $\mathbb{L}$ *is* complete with respect to logging specification LS and execution trace τ *iff* $LS(\tau) \leq \lfloor \mathbb{L} \rfloor$.

Figure 2.2 illustrates graphically the relations of soundness and completeness of audit logs with respect to the semantics of logging.

Relation to Log Queries

As discussed in Section 1.2.1, we make a distinction between logging specifications such as LS_H which define how to record logs, and log queries such as LQ_H which ask questions of logs, and our notions of soundness and completeness apply strictly to logging specifications. However, any logging query must assume a logging specification semantics, hence a log that is demonstrably sound and complete provides the same answers on a given query that an "ideal" log would. This is an important property that is discussed in previous work, e.g. as "sufficiency" in [73].

2.4 Correct Logging Instrumentation is a Safety Property

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

Definition 2.4.1 *For all logging specifications LS, the trace τ is* ideally instrumented for *LS iff for all finite prefixes σ of τ we have $\sigma \rightsquigarrow \mathbb{L}$ where $\mathbb{L}$ is sound and complete with respect to LS and σ.*

We observe that the restriction imposed on logging specifications by Definition 2.2.1, implies that ideal instrumentation of any logging specification is a safety property in the sense defined by Schneider [1].

Theorem 2.4.1 *For all logging specifications LS, the set of ideally instrumented traces is a safety property.*

Proof. If τ is ideally instrumented for LS, then it is prefix-closed by definition. Furthermore, if τ is not ideally instrumented for LS, then it will definitely be rejected in a finite amount of time, since any information in $LS(\tau)$ is encountered after execution of a finite prefix σ of τ by Definition 2.2.1. These two facts obtain the result. $\square$

This result implies that e.g. edit automata can be used to enforce instrumentation of logging specifications (see Section 3.2.2). However, theory related to safety properties and their enforcement by execution monitors [1, 74] do not provide an adequate semantic foundation for audit log generation, nor an account of soundness and completeness of audit logs.

2.5 Implementing Logging Specifications with Program Rewriting

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

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

Ideally, a rewriting algorithm should preserve the semantics of the program it instruments. That is, $\mathcal{R}$ is semantics-preserving if the rewritten program simulates the semantics of the source code, modulo logging steps. We assume given a correspondence relation $:\approx$ on execution traces. A coherent definition of correspondence should be similar to a bisimulation, but it is not necessarily symmetric nor a bisimulation, since the instrumented target program may be in a different language than the source program. We deliberately leave the correspondence relation underspecified, as its definition will depend on the instantiation of the model. Possible definitions are that traces produce the same final value, or that traces when restricted to a set of memory locations are equivalent up to stuttering. We provide an explicit definition of correspondence for FJ-calculus source and target languages in Section 3.2.

Definition 2.5.1 *Rewriting algorithm $\mathcal{R}$ is* semantics preserving *iff for all programs $\mathfrak{p}$ and logging specifications LS such that $\mathcal{R}(\mathfrak{p}, LS)$ is defined, all of the following hold:*

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

In addition to preserving program semantics, a correctly rewritten program constructs a log in accordance with the given logging specification. More precisely, if

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

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' contains the same information as $LS(\tau)$, or at least a sound approximation. Some definitions of $:\approx$ may allow several target-language traces to correspond to source-language traces (as for example in Section 3.2, Definition 4.4.8). In any case, we expect that at least one simulation exists. Hence we write $\mathit{simlogs}(\mathfrak{p}, \tau)$ to denote a nonempty set of logs $\mathbb{L}$ such that, given source language trace τ and target program $\mathfrak{p}$, there exists some trace τ' where $\mathfrak{p} \Downarrow \tau'$ and $\tau :\approx \tau'$ and $\tau' \rightsquigarrow \mathbb{L}$. The name *simlogs* evokes the relation to logs resulting from simulating executions in the target language.

The following definitions then establish correctness conditions for rewriting algorithms. Note that satisfaction of either of these conditions only implies condition (i) of Definition 2.5.1, not condition (ii), so semantics preservation is an independent condition.

Definition 2.5.2 *Rewriting algorithm $\mathcal{R}$ is* sound *iff for all programs $\mathfrak{p}$, logging specifications LS, and finite traces τ where $\mathfrak{p} \Downarrow \tau$, for all $\mathbb{L} \in \mathit{simlogs}(\mathcal{R}(\mathfrak{p}, LS), \tau)$ it is the case that $\mathbb{L}$ is sound with respect to LS and τ.*

Definition 2.5.3 *Rewriting algorithm $\mathcal{R}$ is* complete *iff for all programs $\mathfrak{p}$, logging specifications LS, and finite traces τ where $\mathfrak{p} \Downarrow \tau$, for all $\mathbb{L} \in \mathit{simlogs}(\mathcal{R}(\mathfrak{p}, LS), \tau)$ it is the case that $\mathbb{L}$ is complete with respect to LS and τ.*

2.6 Languages for Logging Specifications

In this section, we go into more detail about information algebra and why it is a good foundation for logging specifications and semantics. We use the formalism of information algebras to characterize and compare the information contained in an audit log with the information contained in an actual execution. In particular, Definition 2.6.5 formally defines logging specification using the operators of information algebra. Example 2.6.1 gives an instance of such logging specification. We follow the proposed formalism in this section to implement audit logging in health informatics.

Various approaches are taken to audit log generation and representation, including logical [18], database [47], and probabilistic approaches [75]. Information algebra is sufficiently general to contain relevant systems as instances, so our notions of soundness and completeness can apply broadly. Here we discuss logical and database approaches.

2.6.1 First Order Logic (FOL)

Logics have been used in several well-developed auditing systems [40, 43], for the encoding of both audit logs and queries. FOL in particular is attractive due to readily available implementation support, e.g. Datalog and Prolog.

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

The properties given in the following Lemma are stated without proof since they are self-evident properties of FOL deduction.

Lemma 2.6.1 *Each of the following properties hold:*

1. $X \vdash \phi$ *for each* $\phi \in X$

2. *If* $X \vdash \phi$ *for each* ϕ *in* Y *and* $Y \vdash \psi$*, then* $X \vdash \psi$

Elements of our algebra are sets of formulas closed under logical entailment. Intuitively, given a set of formulas X, the closure of X is the set of formulas that are logically entailed by X, and thus represents all the information contained in X. In spirit, we follow the treatment of sentential logic as an information algebra explored in related foundational work [13], however our definition of closure is syntactic, not semantic.

Definition 2.6.1 *We define a closure operation* C*, and a set* Φ_{FOL} *of closed sets of formulas:*

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

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

Due to the definition of preconditions, we will be particularly interested in proving properties of sets $C_L(X)$:

Definition 2.6.2 *For each sublanguage* $L \in \mathcal{S}$*, we define closure operator* $C_L(X)$*:*

$$C_L(X) = C(X) \cap L.$$

An important point about such sets is that their closures contain tautological assertions, which may involve predicates P which are not included in L. However, in tautological assertions any predicate does as well as any other, which is an important fact to get hold of for our proofs. Thus we will identify a "dummy" predicate that, in essence, allows us to treat a canonical form of tautologies.

Definition 2.6.3 *We reserve a unary* dummy *predicate D with a countably infinite domain of constants $\mathbf{c}$, and posit an injective function from distinct concrete assertions $P(\bar{\mathbf{c}})$ to distinct $D(\mathbf{c})$. We further define $norm_L(\phi)$ to be the formula ϕ' which is the same as ϕ, but where each $P(\bar{\mathbf{c}}) \notin L$ is replaced with its corresponding image $D(\mathbf{c})$ in the injection. The pointwise extension of $norm_L$ to sets X is denoted $norm_L(X)$.*

Now, we demonstrate canonical forms:

Lemma 2.6.2 *$X \cap L \cup Y \vdash \phi$ iff $X \cap L \cup norm_L(Y) \vdash norm_L(\phi)$.*

Proof. First we prove the left-to-right implication by induction on the derivation of $C(X) \cap L \cup Y \vdash \phi$ and case analysis on the last step in the derivation.

Case Axiom. In this case $\phi \in X \cap L \cup Y$, so either $\phi \in Y$, or $\phi \in X \cap L$. In the former subcase, $norm_L(\phi) \in norm_L(Y)$ by definition and $norm_L(\phi) = \phi$ in the latter subcase. In either subcase, the result holds axiomatically.

Case $\rightarrow$ elimination. In this case we have:

$$\frac{X \cap L \cup Y \vdash \psi \rightarrow \phi \qquad X \cap L \cup Y \vdash \psi}{X \cap L \cup Y \vdash \phi}$$

By the induction hypothesis we have:

$$X \cap L \cup norm_L(Y) \vdash norm_L(\psi \rightarrow \phi) \qquad X \cap L \cup norm_L(Y) \vdash norm_L(\psi)$$

But $norm_L(\psi \to \phi) = norm_L(\psi) \to norm_L(\psi)$ by definition, so the result follows in this case by modus ponens. $\square$

Let *Preds* be the set of all predicate symbols, and let $S \subseteq \mathit{Preds}$ be a set of predicate symbols. We define *sublanguage* L_S to be the set of well-formed formulas over predicate symbols in S (and including boolean atoms T and F, 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 $\mathcal{F} = L_{\mathit{Preds}}$ as the top element.

Lemma 2.6.2 shows that wlog we can modify $\mathcal{S}$ so that every $L \in \mathcal{S}$ contains D. Hence we have immediately:

Lemma 2.6.3 *For all $L, M \in \mathcal{S}$ and $\phi \in \mathcal{F}$, $norm_L(\phi) \in L$, and if $\phi \in M$ then $norm_L(\phi) \in M$.*

This allows us to prove the following important auxiliary results about closures.

Lemma 2.6.4 *If $C(X) \cap L \vdash \phi$ and $\phi \in M$, then $C(X) \cap L \cap M \vdash \phi$.*

Proof. By Lemma 2.6.3 we have $norm_L(\phi) \in L$, and by Lemma 2.6.2 we have $C(X) \cap L \vdash norm_L(\phi)$, hence $norm_L(\phi) \in C(X) \cap L$. But also $norm_L(\phi) \in M$ by assumption and Lemma 2.6.3, so also $norm_L(\phi) \in C(X) \cap L \cap M$. Hence $C(X) \cap L \cap M \vdash norm_L(\phi)$ as an axiom, therefore the result follows by Lemma 2.6.2. $\square$

Lemma 2.6.5 *If $C_L(C_M(X)) \cup Y \vdash \phi$ then $C_{M \cap L}(X) \cup Y \vdash \phi$.*

Proof. The result follows by induction on $C_L(C_M(X)) \cup Y \vdash \phi$ and case analysis on the last step in the derivation. Most cases follow in a straightforward manner; the presence of Y in the formulation is to allow for additional hypotheses, as for example in the case of $\rightarrow$ introduction, as follows.

Case $\rightarrow$ introduction. In this case $\phi = \phi_1 \rightarrow \phi_2$, and we have:

$$\frac{C_L(C_M(X)) \cup Y \cup \{\phi_1\} \vdash \phi_2}{C_L(C_M(X)) \cup Y \vdash \phi_1 \rightarrow \phi_2}$$

but then by the induction hypothesis the judgement $C_{M \cap L}(X) \cup Y \cup \{\phi_1\} \vdash \phi$ is derivable, so the result follows by $\rightarrow$ introduction.

The interesting case is the axiomatic one, i.e. where $\phi \in C_L(C_M(X)) \cup Y$, and specifically the subcase where $\phi \in C_L(C_M(X))$, which follows by Lemma 2.6.4. $\square$

The following Lemma completes the necessary preconditions to prove that the construction Φ_{FOL} is a "domain-free" information algebra [13].

Lemma 2.6.6 *Each of the following properties hold:*

1. *If $X \subseteq Y$ then $C(X) \subseteq C(Y)$.*

2. $C(X \cup Y) = C(X \cup C(Y))$

3. $C(C_L(C_M(X))) = C(C_{M \cap L}(X))$

   for $X \subseteq \mathcal{F}$ and $L, M \in \mathcal{S}$

4. $C_L(C_L(X) \cup Y) = C_L(C_L(X) \cup C_L(Y))$

Proof. Properties (1) and (2) are a consequence of Lemma 2.6.1 as demostrated in [13].

Proof of (3). By definition:

$$C(C_L(C_M(X))) = C(C(C(X) \cap M) \cap L) \qquad C(C_{M \cap L}(X)) = C(C(X) \cap M \cap L)$$

Since $C(X) \cap M \subseteq C(C(X) \cap M)$ by Lemma 2.6.1 property (1), therefore by property (1) in the current Lemma:

$$C(C(X) \cap M \cap L) \subseteq C(C(C(X) \cap M) \cap L).$$

It thus remains to show that:

$$C(C(C(X) \cap M) \cap L) \subseteq C(C(X) \cap M \cap L)$$

which follows by definition of closure and an application of Lemma 2.6.5, taking $Y = \varnothing$ in that Lemma.

Proof of (4). By (2), we have:

$$C(C_L(X) \cup Y) = C(C_L(X) \cup C(Y))$$

and thus also:

$$C(C_L(X) \cup (C(Y) \cap L)) \subseteq C(C_L(X) \cup C(Y))$$

which establishes:

$$C_L(C_L(X) \cup C_L(Y)) \subseteq C_L(C_L(X) \cup Y).$$

For brevity in the remaining, define:

$$A = C_L(C_L(X) \cup C(Y)) \qquad B = C_L(C_L(X) \cup C_L(Y)).$$

To prove the result it now suffices to establish that $A \subseteq B$, so we assume on the contrary that there exists some $\phi \in L$ where $A \vdash \phi$ but $B \not\vdash \phi$. Now, clearly $\phi \notin C(X)$ and $\phi \notin C(Y)$, since in these cases it must be that $B \vdash \phi$ holds. Therefore there exist some minimal nonempty subsets $C \subseteq C(Y)$ and $D \subseteq C_L(X)$ such that $C \cup D \vdash \phi$. Let ψ_D be the conjunction of terms in D. Clearly $\psi_D \in L$. Furthermore, by properties of logic, $C \vdash \psi_D \to \phi$ holds, so that $\psi_D \to \phi \in C(Y)$, and since $\psi_D \in L$ and $\phi \in L$ by construction and assumption, therefore $\psi_D \to \phi \in L$, hence $\psi_D \to \phi \in C_L(Y)$. But $\psi_D \in C_L(X)$ necesarily, so $C_L(X) \cup C_L(Y) \vdash \phi$ by modus ponens and (1). Thus $B \vdash \phi$ also by (1), which is a contradiction, etc. $\square$

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

Definition 2.6.4 *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* $X, Y \in \Phi_{FOL}$

These definitions of focusing and combination enjoy a number of properties within the algebra, as stated in the following Theorem, establishing that the construction is an information algebra. FOL has been treated as an information algebra before, but our definitions of combination and focusing and hence the result are novel.

Theorem 2.6.1 *Structure* $(\Phi_{FOL}, \mathcal{S})$ *with focus operation* $X^{\Rightarrow S}$ *and combination operation* $X \otimes Y$ *forms a* domain-free information algebra*.*

Proof. The following properties hold immediately according to Lemma 2.6.6 and Lemma 2.6.1, and thus $(\Phi_{FOL}, \mathcal{S})$ is an information algebra [13]:

- *Semigroup*: Φ is associative and commutative under combination, and $C(\varnothing)$ is a neutral element with $X \otimes C(\varnothing) = X$ for all $X \in \Phi$.
- *Transitivity*: $\left(X^{\Rightarrow L}\right)^{\Rightarrow M} = X^{\Rightarrow L \cap M}$ for all $X \in \Phi$ and $L, M \in \mathcal{S}$.
- *Combination*: $\left(X^{\Rightarrow L} \otimes Y\right)^{\Rightarrow L} = X^{\Rightarrow L} \otimes Y^{\Rightarrow L}$ for all $X, Y \in \Phi$ and $L \in \mathcal{S}$.
- *Support*: $X^{\Rightarrow \mathcal{F}} = X$ for all $X \in \Phi$.
- *Idempotence*: $X \otimes X^{\Rightarrow L} = X$ for all $X \in \Phi$ and $L \in \mathcal{S}$.

□

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

Example 2.6.1 *We return to the example described in Section 1.2.1 to show how FOL can express break the glass logging specifications. Adapting a logic programming style, the trace of a program can be viewed as a fact base, and the logging specification LS_H performs resolution of a* LoggedCall *predicate, defined via the following Horn clause we call ψ_H:*

$$\begin{aligned}\forall t, d, s, u.(\mathrm{Call}(t, \mathbf{read}, u, d) \wedge \mathrm{Call}(s, \mathbf{breakGlass}, u) \wedge s < t\ \wedge \mathrm{PatientInfo}(d))\\ \implies \mathrm{LoggedCall}(t, \mathbf{read}, u, d)\end{aligned}$$

Here we imagine that **breakGlass** *is a break the glass function where u identifies the current user and* PatientInfo *is a predicate specifying which files contain patient information. The log contains only valid instances of* LoggedCall *given a particular trace, which specify the user and sensitive information accessed following glass breaking, which otherwise would be disallowed by a separate access control policy.*

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

Definition 2.6.5 *The function spec is given a pair (X, S) and returns a* FOL logging specification*, i.e. a function from traces to elements of* Φ_{FOL}*:*

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

In any logging specification $spec(X,S)$*, we call* X *the* guidelines.

The above example LS_H would then be formally defined as $spec(\psi_H, \{\text{LoggedCall}\})$.

2.6.2 Relational Database

As discussed in Section 2.1.1, relational algebra is a canonical example of an information algebra, which provides information algebraic analysis of relational databases. However, that formulation is of little use in case we seek a formal relation between recorded logs in the relational database and the semantics of logging specification according to Definition 2.6.5. Thus, in this section, we formulate a novel instantiation of relational databases which could be leveraged for correct audit purposes. We define databases D as sets of relations, where a relation X is a set of *tuples* f. We write $((a_1 : x_1), ..., (a_n : x_1))$ to denote an n-ary tuple with attributes (aka label) a_i associated with values x_i. Databases are elements of the information algebra, and sublanguages S are collections of sets of attributes. Each set of attributes corresponds to a specific relation. Focusing is the restriction to particular relations in a database, and combination is the union of databases. Hence, letting $\leq_{RA}$ denote the relational algebra information ordering, $D_1 \leq_{RA} D_2$ iff $D_1 \otimes D_2 = D_2$. We refer to this algebra as Φ_{RA}. In this context, a trace can be interpreted as a collection of relations, and logging specifications can be defined using selects. Relational databases are also heavily used for storing and querying audit logs.

Let $\mathcal{A}$ be a denumerable set of attribute names. Moreover, let $\mathbb{R}$ be the universe for relations, i.e., $\mathbb{R} = \{R \subseteq A_{a_1} \times \cdots A_{a_m} \mid a_i \in \mathcal{A}\}$. Note that A_{a_i} is the domain of values for attribute a_i. We denote the arity of a relation R with $arity(R)$.

Definition 2.6.6 *Let $Name : \mathbb{R} \to \mathcal{P}(\mathcal{A})$ be defined as $Name(R) = \{a_1, \cdots, a_{arity(R)}\}$, if $R \subseteq A_{a_1} \times \cdots A_{a_{arity(R)}}$.*

Definition 2.6.7 *Let database D be a finite subset of $\mathbb{R}$ containing finite relations, i.e., a database D is a finite collection of relations $R \in \mathbb{R}$, where each R is a finite set of tuples defining the relation. Φ_{RA} is defined as the set of all databases.*

We also define the querying sublanguages as the sets of relation names, i.e., $S \in \mathcal{P}(\mathcal{P}(\mathcal{A}))$. Next, we define the information algebra operations:

Definition 2.6.8 *Define:*

- *Focusing: $D^{\Rightarrow S} = \{R \in D \mid Name(R) \in S\}$, where $D \in \Phi_{RA}$, and $S \in \mathcal{P}(\mathcal{P}(\mathcal{A}))$ and finite,*
- *Combination: $D_1 \otimes D_2 = \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\}$.*

Note that in case some relation is not defined in a database, we assume it is defined as an empty relation. We also define a mapping which represents non-trivial relation names in a database:

Definition 2.6.9 *Let $Names : \Phi_{RA} \to \mathcal{P}(\mathcal{P}(\mathcal{A}))$ be defined as*

$$Names(D) = \{Name(R) \mid R \in D, R \neq \emptyset\}.$$

In what follows we show that $(\Phi_{RA}, \mathcal{P}(\mathcal{P}(\mathcal{A})))$ is an information algebra.

Lemma 2.6.7 *Φ_{RA} is a commutative semigroup.*

Proof. We need to show that

- Φ_{RA} is associative on combination:

  Holds straightforwardly based on the associativity of union on sets:

$$\begin{aligned}
D_1 \otimes (D_2 \otimes D_3) &= D_1 \otimes \{R_2 \cup R_3 \mid R_i \in D_i, i \in \{2,3\}, \\
&\qquad\qquad Name(R_2) = Name(R_3)\} \\
&= \{R_1 \cup (R_2 \cup R_3) \mid R_i \in D_i, i \in \{1,2,3\}, \\
&\qquad\qquad Name(R_1) = Name(R_2) = Name(R_3)\} \\
&= \{(R_1 \cup R_2) \cup R_3 \mid R_i \in D_i, i \in \{1,2,3\}, \\
&\qquad\qquad Name(R_1) = Name(R_2) = Name(R_3)\} \\
&= \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1,2\}, \\
&\qquad\qquad Name(R_1) = Name(R_2)\} \otimes D_3 \\
&= (D_1 \otimes D_2) \otimes D_3.
\end{aligned}$$

- Φ_{RA} is commutative on combination:

  Holds straightforwardly based on the commutativity of union on sets:

$$\begin{aligned}
D_1 \otimes D_2 &= \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\} \\
&= \{R_2 \cup R_1 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\} \\
&= D_2 \otimes D_1.
\end{aligned}$$

- There exists a neutral element I such that for all D, $D \otimes I = D$:

Let $I = \{\emptyset\}$. Obviously, $D \otimes I = D$ as $\emptyset$ is the neutral element for union.

$\square$

Lemma 2.6.8 *Transitivity:* $(D^{\Rightarrow S_1})^{\Rightarrow S_2} = D^{\Rightarrow S_1 \cap S_2}$.

Proof.

$$\begin{aligned}
(D^{\Rightarrow S_1})^{\Rightarrow S_2} &= \{R \in D \mid \mathit{Name}(R) \in S_1\}^{\Rightarrow S_2} \\
&= \{R \in D \mid \mathit{Name}(R) \in S_1, \mathit{Name}(R) \in S_2\} \\
&= \{R \in D \mid \mathit{Name}(R) \in S_1 \cap S_2\} \\
&= D^{\Rightarrow S_1 \cap S_2}.
\end{aligned}$$

$\square$

Lemma 2.6.9 *Combination:* $({D_1}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = {D_1}^{\Rightarrow S} \otimes {D_2}^{\Rightarrow S}$.

Proof.

$$\begin{aligned}
({D_1}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} &= (\{R_1 \in D \mid \mathit{Name}(R) \in S\} \otimes D_2)^{\Rightarrow S} \\
&= \{R_1 \cup R_2 \mid R_1 \in D_1, \mathit{Name}(R_1) \in S, R_2 \in D_2, \\
&\qquad \mathit{Name}(R_1) = \mathit{Name}(R_2)\}^{\Rightarrow S} \\
&= \{R_1 \cup R_2 \mid R_1 \in D_1, \mathit{Name}(R_1) \in S, R_2 \in D_2, \\
&\qquad \mathit{Name}(R_1) = \mathit{Name}(R_2), \mathit{Name}(R_1 \cup R_2) \in S\}.
\end{aligned}$$

Obviously, if $\mathit{Name}(R_1) = \mathit{Name}(R_2)$, then $\mathit{Name}(R_1 \cup R_2) = \mathit{Name}(R_1)$. We thus have

$$(D_1{}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = \{R_1 \cup R_2 \mid R_1 \in D_1, \mathit{Name}(R_1) \in S, R_2 \in D_2, \\ \mathit{Name}(R_1) = \mathit{Name}(R_2)\}$$

Moreover,

$$\begin{aligned} D_1{}^{\Rightarrow S} \otimes D_2{}^{\Rightarrow S} &= \{R_1 \in D \mid \mathit{Name}(R_1) \in S\} \otimes \{R_2 \in D \mid \mathit{Name}(R_2) \in S\} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, \mathit{Name}(R_1) \in S, R_2 \in D_2, \mathit{Name}(R_2) \in S, \\ &\qquad \mathit{Name}(R_1) = \mathit{Name}(R_2)\} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, \mathit{Name}(R_1) \in S, R_2 \in D_2, \\ &\qquad \mathit{Name}(R_1) = \mathit{Name}(R_2)\}. \end{aligned}$$

Thus, $(D_1{}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = D_1{}^{\Rightarrow S} \otimes D_2{}^{\Rightarrow S}$. □

Lemma 2.6.10 *Support:* $\forall D, \exists S, D^{\Rightarrow S} = D$.

Proof. Let $S = \mathit{Names}(D)$. Then, $D^{\Rightarrow \mathit{Names}(D)} = D$. □

Lemma 2.6.11 *Idempotence:* $D \otimes D^{\Rightarrow S} = D$.

Proof.

$$\begin{aligned} D \otimes D^{\Rightarrow S} &= \{R \mid R \in D\} \otimes \{R \in D \mid \mathit{Name}(R) \in S\} \\ &= \{R \in D \mid \mathit{Name}(R) \notin S\} \cup \{R \in D \mid \mathit{Name}(R) \in S\} \\ &= D. \end{aligned}$$

□

Corollary 2.6.1 $(\Phi_{RA}, \mathcal{P}(\mathcal{P}(\mathcal{A})))$ *is an information algebra.*

2.6.3 Transforming and Combining Audit Logs

Multiple audit logs from different sources are often combined in practice. Also, logging information is often transformed for storage and communication. For example, log data may be generated in common event format (CEF), which is parsed and stored in relational database tables, and subsequently exported and communicated via JSON. In all cases, it is necessary to characterize the effect of transformation (if any) on log information, and relate queries on various representations to the logging specification semantics. Otherwise, it is unclear what is the relation of log queries to log-generating programs.

To address this, information algebra provides a useful concept called *monotone mapping*. Given two information algebras Ψ_1 and Ψ_2 with ordering relations $\leq_1$ and $\leq_2$ respectively, a mapping μ from elements X, Y of Ψ_1 to elements $\mu(X), \mu(Y)$ of Ψ_2 is monotone iff $X \leq_1 Y$ implies $\mu(X) \leq_2 \mu(Y)$. For example, assuming that Ψ_1 is our FOL information algebra while Ψ_2 is relational algebra, we can define a monotone mapping using a *least Herbrand interpretation* [76], denoted $\mathfrak{H}$, and by positing a function *attrs* from n-ary predicate symbols to functions mapping numbers $1, ..., n$ to labels. That is, *attrs*(P)(n) is the label associated with the nth argument of predicate P. We require that if $\mathrm{P} \neq \mathrm{Q}$ then $attrs(\mathrm{P})(j) \neq attrs(\mathrm{Q})(k)$ for all j, k. To map predicates to tuples we have:

$$tuple(\mathrm{P}(x_1, \ldots, x_n)) = ((attrs(\mathrm{P})(1) : x_1), \ldots, (attrs(\mathrm{P})(n) : x_n))$$

Then to obtain a relation from all valid instances of a particular predicate P given formulas X we define:

$$R_{\mathrm{P}}(X) = \{\mathit{tuple}(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(X)\}$$

Now we define the function *rel* which is collection of all relations obtained from X, where $\mathrm{P}_1, ..., \mathrm{P}_n$ are the predicate symbols occurring in X:

$$\mathit{rel}(X) = \{R_{\mathrm{P}_1}(X), \cdots, R_{\mathrm{P}_n}(X)\}$$

Theorem 2.6.2 *rel is a monotone mapping.*

Proof. We need to show that $X \leq_{FOL} Y$ implies $\mathit{rel}(X) \leq_{RA} \mathit{rel}(Y)$.
From $X \leq_{FOL} Y$ we have $\mathfrak{H}(X) \subseteq \mathfrak{H}(Y)$. Let $R \in \mathit{rel}(X)$ and $R' \in \mathit{rel}(Y)$, such that $\mathit{Name}(R) = \mathit{Name}(R')$. Then, $R = R_{\mathrm{P}}(X)$ and $R' = R_{\mathrm{P}}(Y)$, for some n-ary predicate symbol P such that $\mathit{Name}(R) = \{\mathit{attrs}(\mathrm{P})(1), \cdots, \mathit{attrs}(\mathrm{P})(n)\}$. Since $\mathfrak{H}(X) \subseteq \mathfrak{H}(Y)$,

$$\begin{aligned} R &= \{\mathit{tuple}(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(X)\} \\ &\subseteq \{\mathit{tuple}(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(Y)\} = R'. \end{aligned}$$

Therefore, $R \cup R' = R'$. Then,

$$\begin{aligned} \mathit{rel}(X) \otimes \mathit{rel}(Y) &= \{R \cup R' \mid R \in \mathit{rel}(X), R' \in \mathit{rel}(Y), \mathit{Name}(R) = \mathit{Name}(R')\} \\ &= \{R' \mid R \in \mathit{rel}(X), R' \in \mathit{rel}(Y), \mathit{Name}(R) = \mathit{Name}(R')\} \\ &= \mathit{rel}(Y). \end{aligned}$$

This implies $rel(X) \leq_{RA} rel(Y)$ by information containment definition. $\square$

Thus, if we wish to generate an audit log $\mathbb{L}$ as a set of FOL formulas, but ultimately store the data in a relational database, we are still able to maintain a formal relation between stored logs and the semantics of a given trace τ and specification LS. E.g., if a log $\mathbb{L}$ is sound with respect to τ and LS, then $rel(\lfloor\mathbb{L}\rfloor) \leq_{RA} rel(LS(\tau))$. While the data in $rel(\lfloor\mathbb{L}\rfloor)$ may very well be broken up into multiple relations in practice, e.g. to compress data and/or for query optimization, the formalism also establishes correctness conditions for the transformation that relate resulting information to the logging semantics $LS(\tau)$ by way of the mapping. We reify this idea in our OpenMRS implementation as discussed in Section 3.3.2.

Chapter 3

Temporal properties of function calls: Break the glass policies

In this chapter, we specify our language model, a core OO calculus in Section 3.1. Next, we specify a class of logging specifications whose concern is the temporal properties of function calls [77]. This class, called **Calls** is introduced in Section 3.2.1. Section 3.2.2 proposes an edit automata to enforce ideal instrumentation. We develop a provably correct enforcement mechanism by program instrumentation for **Calls** in Section 3.2.3 and Section 3.2.4. We also discuss our case study in health informatics in Section 3.3. In Section 3.4, we explore the techniques to avoid memory leakage in the deployment of **Calls**.

3.1 Source Language

In this section, we define the source language which is the basis for our language model in this dissertation. It includes the definitions of configurations and execution traces.function $toFOL(\cdot)$ that shows how we concretely model execution traces in FOL. This language model which is used with potential minor modifications in the following sections for studying temporal properties of function calls and dynamic integrity taint analysis.

We begin the technical presentation with definition of our language model based on Featherweight Java (FJ) [27]. FJ is a core calculus that includes class hierarchy definitions, subtyping, dynamic dispatch, and other basic features of Java. Language FJ is a simple call-by-value OO core calculus with functional flavor. An FJ program is a pair $(\mathtt{e}, CT)$ where $\mathtt{e}$ is an expression, and CT is a *class table* which maintains class definitions. An FJ configuration is a pair $(\mathtt{e}, n)$ of an expression $\mathtt{e}$ and a timestamp n. We assume the implicit existence of class tables as a component of configurations, since class tables do not change during program execution.

3.1.1 Syntax

Figure 3.1 demonstrates the syntax for FJ. 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 $\mathtt{C.m(e)}$. This form is used to identify the method $\mathtt{C.m}$ associated with a current evaluation context (aka the "activation frame"). This does not really change the semantics, but is a useful feature for our specification of sanitizer endorsement since return values from sanitizers need to be endorsed– see the Invoke and Return rules in the operational semantics for its usage.

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

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

$$\kappa ::= (\mathtt{e}, n) \qquad \mathfrak{p} ::= (\mathtt{e}, CT)$$

Figure 3.1: FJ Syntax

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

3.1.2 Semantics

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

The class table, field and method body lookup, and inheritance

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. These definitions are given Figure 3.2 and Figure 3.3. We assume a given class table CT during evaluation, which will be clear from context.

Using CT we also define a predicate to denote the inherited methods in a class

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

Figure 3.2: Class Fields in FJ

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

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

Figure 3.3: Method Bodies in FJ

(Figure 3.4). The class table is implicit in this definition.

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. Method type lookup is defined in Figure 3.5. Although we omit FJ type analysis from this presentation, method type lookup will be useful for taint analysis instrumentation (Definition 4.3.1).

Operational semantics

Now, we can define the operational semantics of FJ. We define these as a small step relation in the usual manner, depicted in Figure 3.6.

We use $\rightarrow^*$ to denote the reflexive, transitive closure of $\rightarrow$. 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 $\kappa_i \rightarrow \kappa_{i+1}$ for $0 < i < n$. Note that an execution trace τ may

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

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

Figure 3.4: Method Inheritance in FJ

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

Figure 3.5: Method Types in FJ

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 $\mathfrak{p}(\theta)$ where θ is an object, we assume that all class tables CT include an entry point `TopLevel.main`, where `TopLevel` objects have no fields. We define $\mathfrak{p}(\theta) = \mathtt{new\ TopLevel().main}(\theta)$, and we write $\mathfrak{p}(\theta) \Downarrow \tau$ iff trace τ begins with the configuration $(\mathfrak{p}(\theta), 0)$.

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

Context

$$\dfrac{(\texttt{e}, n) \to (\texttt{e}', n')}{(\texttt{E}[\texttt{e}], n) \to (\texttt{E}[\texttt{e}'], n')}$$

Field

$$\dfrac{\mathit{fields}_{CT}(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}} \qquad \texttt{f}_i \in \bar{\texttt{f}}}{(\texttt{new C}(\bar{\texttt{v}}).\texttt{f}_i, n) \to (\texttt{v}_i, n+1)}$$

Invoke

$$\dfrac{\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}}{(\texttt{new C}(\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}}), n) \to (\texttt{C.m}(\texttt{e}[\texttt{new C}(\bar{\texttt{v}})/\texttt{this}][\bar{\texttt{u}}/\bar{\texttt{x}}]), n+1)}$$

Return

$$(\texttt{C.m}(\texttt{v}), n) \to (\texttt{v}, n+1)$$

Figure 3.6: FJ Operational Semantics

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

Figure 3.7: Definition of toFOL(·) for Configurations.

Definition 3.1.1 *We define toFOL(·) as a mapping on traces and configurations:*

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

such that $\mathit{toFOL}(\sigma) = \bigcup_i \mathit{toFOL}(\kappa_i)$ *for* $\sigma = \kappa_1 \cdots \kappa_k$. *We define* $\mathit{toFOL}(\kappa)$ *as in Figure 3.7.*

3.2 Rewriting Programs with Logging Specifications

Since correct logging instrumentation is a safety property (Section 2.4), there are several possible implementation strategies. For example, one could define an edit automata that enforces the property (see Section 3.2.2), that could be implemented either as a separate program monitor or using IRM techniques [14]. But since we are interested in program rewriting for a particular class of logging specifications, the approach we discuss here is more simply stated and proven correct than a general IRM methodology.

We specify a class of logging specifications of interest, along with a program rewriting algorithm that is sound and complete for it. We consider FJ to serve as the formal setting to establish correctness of a program rewriting approach to correct instrumentation of logging specification. We use this same approach to implement an auditing tool for OpenMRS, described in the next section. The supported class of logging specifications is predicated on temporal properties of method calls and characteristics of their arguments. This class has practical potential since security-sensitive operations are often packaged as functions or methods (e.g. in medical records software [78]), and the supported class allows complex policies such as break the glass to be expressed. The language of logging specifications is FOL, and we use Φ_{FOL} to define the semantics of logging and prove correctness of the algorithm.

3.2.1 Specifications Based on Function Call Properties

We define a class **Calls** of logging specifications that capture temporal properties of function calls, such as those reflected in break the glass policies. We restrict specification definitions to safe Horn clauses to ensure applicability of well-known results and total algorithms such as Datalog [76]. Specifications in **Calls** support logging of calls to a specific methods $\mathtt{C}_0.\mathtt{m}_0$ that happen after functions $\mathtt{C}_1.\mathtt{m}_1, \ldots, \mathtt{C}_n.\mathtt{m}_n$

$$\forall t, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}, \mathtt{D} \,.\, \mathrm{Call}(t, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \land \mathrm{Inherit}(\mathtt{m}, \mathtt{C}, \mathtt{D}) \implies \mathrm{Call}(t, \mathtt{D}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$$

$$\forall t_0, \ldots, t_n, \bar{\mathtt{v}}_0, \ldots, \bar{\mathtt{v}}_n, \bar{\mathtt{u}}_0, \ldots, \bar{\mathtt{u}}_n \,.\, \mathrm{Call}(t_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0) \bigwedge_{i=1}^{n} (\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \land t_i < t_0) \land$$
$$\phi((t_0, \bar{\mathtt{v}}_0, \bar{\mathtt{u}}_0), \ldots, (t_n, \bar{\mathtt{v}}_n, \bar{\mathtt{u}}_n)) \implies \mathrm{LoggedCall}(t_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0).$$

Figure 3.8: Horn Clauses in **Calls**

are called. Conditions on all object arguments, method arguments, and times of method invocation, can be defined via a predicate ϕ. Hence more precise requirements can be imposed, e.g. a linear ordering on invocations, particular values of method arguments, etc.

Definition 3.2.1 **Calls** *is the set of all logging specifications spec*$(X, \{\mathrm{LoggedCall}\})$ *where X contains safe Horn clauses of the form given in Figure 3.8.*

While set X may contain other safe Horn clauses, in particular definitions of predicates occurring in ϕ, no other Horn clause in X uses the predicate symbols LoggedCall, Value, Context, Call, ReturnValue, or GetField. For convenience in the following, we define *Logevent*$(LS) = \mathtt{C}_0.\mathtt{m}_0$ and *Triggers*$(LS) = \{\mathtt{C}_1.\mathtt{m}_1, ..., \mathtt{C}_n.\mathtt{m}_n\}$.

We note that specifications in **Calls** clearly satisfy Definition 2.2.1, since preconditions for logging a particular call to $\mathtt{C}_0.\mathtt{m}_0$ must be satisfied at the time of that call.

3.2.2 Edit Automata Enforcement of **Calls** Specifications

Considering Theorem 2.4.1, we observe that given a logging specification in **Calls**, we can easily define an edit automata that enforces this property. The following definition is in the "guarded command" style used by Schneider [1]. The array A is used to store potentially multiple values of potentially multiple calls to each function g_i.

Theorem 3.2.1 *Given spec(X, S) ∈* **Calls***, the ideal instrumentation property is enforced by the edit automata in Figure 3.9. It is defined using the following predicates on input configurations κ:*

$$\begin{aligned} \mathrm{Call}(t, C, \bar{x}, m, \bar{y}) &: \quad \textit{means } \mathrm{Call}(t, C, \bar{x}, m, \bar{y}) \in \textit{toFOL}(\kappa) \\ \phi \textit{ logged} &: \quad \textit{means } \phi \in \textit{toFOL}(\textit{logof}(\kappa)) \end{aligned}$$

and also T and V denote the universes of timestamps and program values respectively.

Proof.

Straightforward by induction on traces and definitions of edit automata [74]. □

However, as indicated in Section 2.4, this technique does not provide an adequate semantic foundation for log generation, and consequently correctness studies. Studying the correctness of audit logging requires the ability to analyze whether the entries are "appropriately" added to the log in each step of computation. That is, we need to be able to judge whether a log entry has been recorded according to the logging

$$
\begin{array}{rrcl}
\textbf{state vars} & \multicolumn{3}{l}{A[n+1] : \textbf{array of sets of } T \times \bar{V} \times \bar{V} \textbf{ initial } \emptyset} \\
\textbf{transitions} & \textbf{not } (\text{Call}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \vee \text{Call}(t_i, \mathtt{C}_i, \bar{x}_i, \mathtt{m}_i, \bar{y}_i)) & \longrightarrow & \textbf{skip} \\
 & \text{Call}(t_1, \mathtt{C}_1, \bar{x}_1, \mathtt{m}_1, \bar{y}_1) & \longrightarrow & A[1] := A[1] \cup \{(t_1, \bar{x}_1, \bar{y}_1)\} \\
 & & \vdots & \\
 & \text{Call}(t_n, \mathtt{C}_n, \bar{x}_n, \mathtt{m}_n, \bar{y}_n) & \longrightarrow & A[n] := A[n] \cup \{(t_n, \bar{x}_n, \bar{y}_n)\} \\
 & \begin{array}{l} \text{Call}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \wedge \\ (\textbf{not}(\exists \ell \in A[1] * \cdots * A[n].\phi((t_0, \bar{x}_0, \bar{y}_0), \ell)) \vee \\ \quad \text{LoggedCall}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \text{ logged}) \end{array} & \longrightarrow & \textbf{skip} \\
\textbf{editing rules} & \begin{array}{l} \text{Call}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \wedge \\ \exists \ell \in A[1] * \cdots * A[n].\phi((t_0, \bar{x}_0, \bar{y}_0), \ell) \wedge \\ \text{LoggedCall}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \text{ not logged} \end{array} & \longrightarrow & \text{add LoggedCall}(t_0, \mathtt{C}_0, \bar{x}_0, \mathtt{m}_0, \bar{y}_0) \text{ to log}
\end{array}
$$

Figure 3.9: Edit Automata to Enforce Ideal Instrumentation

specification (necessity), and also what the logging specification advertises *is* added to the log (sufficiency). However, note that given a logging specification LS, an ideally instrumented trace is defined as a trace whose every prefix generates correct log wrt that prefix and LS (Definition 2.4.1). This definition does not convey the required formal components to study the necessity and sufficiency conditions of logging correctness in each step of computation. In the following we propose an instrumentation approach to enforce audit logging in a provably correct manner.

3.2.3 Target Language

In order to specify a rewriting algorithm for provably correct enforcement of audit logging, we need to establish the target language as the codomain of the rewriting algorithm, i.e., the algorithm rewrites programs into the ones in the target language. The syntax of our target language FJ_{log} extends FJ syntax with a command to track *logging preconditions* ($\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}})$), i.e. calls to logging triggers, and a command

$$\mathtt{e} ::= \ldots \mid \mathtt{this.callEvent}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e} \mid \mathtt{this.emit}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e} \qquad\qquad \kappa ::= (\mathtt{e}, n, X, \mathbb{L})$$

Precondition

$$\frac{}{(\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})\}, \mathbb{L})}$$

Log

$$\frac{X \cup X_{Guidelines} \vdash \mathrm{LoggedCall}(n-1, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})}{(\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}, n, X, \mathbb{L} \cup \{\mathrm{LoggedCall}(n-1, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})\})}$$

NoLog

$$\frac{X \cup X_{Guidelines} \nvdash \mathrm{LoggedCall}(n-1, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})}{(\mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}, n, X, \mathbb{L})}$$

Figure 3.10: $\mathrm{FJ}_{\mathrm{log}}$ *Syntax and Sematics Extensions.*

to emit log entries ($\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}})$). Configurations are extended to include a set X of logging preconditions, and an audit log $\mathbb{L}$. These extensions are depicted in Figure 3.10.

The semantics of $\mathrm{FJ}_{\mathrm{log}}$ extends the semantics of FJ with new rules for methods $\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}})$ and $\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}})$, which update the set of logging preconditions and audit log respectively. An instrumented program uses the set of logging preconditions to determine when it should emit events to the audit log. The semantics is parameterized by a guideline $X_{Guidelines}$, typically taken from a logging specification. Given the definition of **Calls**, these semantics would be easy to implement using e.g. a Datalog proof engine. The extensions to $\mathrm{FJ}_{\mathrm{log}}$ semantics is also given in Figure 3.10.

Note that to ensure that these instrumentation commands do not change execution behavior, the configuration's time is not incremented when $\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}})$ and $\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}})$ are evaluated. That is, the configuration time counts the number of source language computation steps.

The rules Log and NoLog rely on checking whether $X_{Guidelines}$ and logging preconditions X entail LoggedCall$(n-1, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$.

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 special cases for when the source execution is about to perform a function application that the target execution will track or log via an $\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}})$ or $\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}})$ command. In these cases, the target execution may be ahead by one or two steps, allowing time for addition of information to the log.

Definition 3.2.2 *Given source language execution trace $\tau = \kappa_0 \dots \kappa_m$ and target language execution trace $\tau' = \kappa'_0 \dots \kappa'_n$, where $\kappa_i = (\mathtt{e}_i, t_i)$ and $\kappa'_i = (\mathtt{e}'_i, t'_i, X_i, \mathbb{L}_i)$, $\tau :\approx \tau'$ iff $\mathtt{e}_0 = \mathtt{e}'_0$ and either*

1. $\mathtt{e}_m = \mathtt{e}'_n$ *(taking $=$ to mean syntactic equivalence); or*

2. $\mathtt{e}_m = \mathtt{e}'_{n-1}$ *and* $\mathtt{e}'_n = \mathtt{v.callEvent}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e}'$ *for some* $\mathtt{v}$*,* $\mathtt{m}$*,* $\bar{\mathtt{u}}$*, and* $\mathtt{e}'$*; or*

3. $\mathtt{e}_m = \mathtt{e}'_{n-2}$ *and* $\mathtt{e}'_n = \mathtt{v.emit}(\mathtt{m}, \bar{\mathtt{u}}); \mathtt{e}'$ *for some* $\mathtt{v}$*,* $\mathtt{m}$*,* $\bar{\mathtt{u}}$*, and* $\mathtt{e}'$*.*

Finally, we need to define *toFOL*$(\mathbb{L})$ for audit logs $\mathbb{L}$ produced by an instrumented program. Since our audit logs are just sets of formulas of the form LoggedCall$(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$, we define *toFOL*$(\mathbb{L}) = \mathbb{L}$.

3.2.4 Program Rewriting Algorithm

Our program rewriting algorithm $\mathcal{R}_{\text{FJ}}$ takes an FJ program $\mathfrak{p} = (\mathtt{e}, CT)$, a logging specification $LS = spec(X_{Guidelines}, \{\text{LoggedCall}\}) \in$ **Calls**, and produces an FJ$_{\text{log}}$ program $\mathfrak{p}' = (\mathtt{e}', CT')$ such that $\mathtt{e}$ and $\mathtt{e}'$ are identical, and CT' is identical to CT

$$\frac{\texttt{C.m} = \mathit{Logevent}(\mathit{LS}) \qquad \mathit{mbody}_{CT}(\texttt{m},\texttt{C}) = \bar{\texttt{x}},\texttt{e}}{\mathit{mbody}_{\mathcal{R}(CT)}(\texttt{m},\texttt{C}) = \bar{\texttt{x}}, \texttt{this.callEvent}(\texttt{m},\bar{\texttt{x}});\texttt{this.emit}(\texttt{m},\bar{\texttt{x}});\texttt{e}} \qquad \frac{\texttt{C.m} \in \mathit{Triggers}(\mathit{LS}) \qquad \mathit{mbody}_{CT}(\texttt{m},\texttt{C}) = \bar{\texttt{x}},\texttt{e}}{\mathit{mbody}_{\mathcal{R}(CT)}(\texttt{m},\texttt{C}) = \bar{\texttt{x}}, \texttt{this.callEvent}(\texttt{m},\bar{\texttt{x}});\texttt{e}}$$

$$\frac{\texttt{C.m} \notin \mathit{Triggers}(\mathit{LS}) \cup \{\mathit{Logevent}(\mathit{LS})\}}{\mathit{mbody}_{\mathcal{R}(CT)}(\texttt{m},\texttt{C}) = \mathit{mbody}_{CT}(\texttt{m},\texttt{C})}$$

Figure 3.11: Axioms for Rewriting Algorithm $\mathcal{R}_{\mathrm{FJ}}$

except for the addition of $\texttt{callEvent}(\texttt{m},\bar{\texttt{u}})$ and $\texttt{emit}(\texttt{m},\bar{\texttt{u}})$ method invocations. The algorithm is straightforward: we modify the class table to add $\texttt{callEvent}(\texttt{m},\bar{\texttt{u}})$ to the definition of any method $\texttt{C.m} \in \mathit{Triggers}(\mathit{LS}) \cup \{\mathit{Logevent}(\mathit{LS})\}$ and add $\texttt{emit}(\texttt{m},\bar{\texttt{u}})$ to the definition of method $\texttt{C.m} = \mathit{Logevent}(\mathit{LS})$.

Definition 3.2.3 *For FJ program $\mathfrak{p} = (\texttt{e}, CT)$ and logging specifications $LS \in$* **Calls**, *define:*

$$\mathcal{R}_{\mathrm{FJ}}((\texttt{e}, CT), LS) = (\texttt{e}, \mathcal{R}(CT))$$

where $\mathcal{R}(CT)$ is the smallest class table satisfying the axioms given in Figure 3.11.

Program rewriting algorithm $\mathcal{R}_{\mathrm{FJ}}$ is semantics preserving, sound, and complete for **Calls**. We have completely formalized these results (modulo well-known Horn clause logic definitions and properties) in Coq [79]. In this section we summarize our results.

Theorem 3.2.2 *Program rewriting algorithm $\mathcal{R}_{\mathrm{FJ}}$ is semantics preserving (Definition 2.5.1).*

Proof. Intuitively, the addition of $\texttt{this.callEvent}(\texttt{m},\bar{\texttt{v}})$ and $\texttt{this.emit}(\texttt{m},\bar{\texttt{v}})$ methods does not interfere with FJ evaluation. The proof follows easily by induction on the number of small-step reductions of programs. $\square$

Our proof strategy for soundness and completeness of $\mathcal{R}_{\mathrm{FJ}}$ is to show that an audit log produced by an instrumented program is the refinement of the least Herbrand model of the logging specification semantics unioned with the logging specification's guidelines. By showing that audit logs combined with the guidelines are the least Herbrand models of the logging specification semantics, we show that they contain the same information. This implies soundness and completeness of program rewriting.

The following Lemma relates the syntactic property of closure with the properties of a least Herbrand model [76, 80], and shows that the least Herbrand model of X contains the same information as X. It holds by the soundness and completeness of the logic.

Lemma 3.2.1 $C(\mathfrak{H}(X)) = C(X)$ *and* $\mathfrak{H}(X) = \mathfrak{H}(C(X))$.

The following Lemmas states a similar but subtly different property relevant to sub-language focusing that we will use in Theorem 3.2.3.

Lemma 3.2.2 $C(C(\mathfrak{H}(X)) \cap L) = C(\mathfrak{H}(X) \cap L)$.

The key idea underlying the soundness of the program rewriting algorithm is that any facts that are added to the set of logging preconditions or the audit log during execution of the instrumented program are true facts: they are in the model of the corresponding source language execution trace.

Lemma 3.2.3 *Let* $\mathfrak{p}$ *be a* FJ *program and* $LS \in \mathbf{Calls}$ *be a logging specification. For all target language execution traces* τ *such that* $\mathcal{R}_{\mathrm{FJ}}(\mathfrak{p}, LS) \Downarrow \tau$*, where* $\tau = \kappa_0 \ldots \kappa_n$ *and* $\kappa_n = (\mathtt{e}, t, X, \mathbb{L})$*, there exists a source language execution trace* τ' *such that* $\tau' :\approx \tau$ *and* $\mathfrak{p} \Downarrow \tau'$ *and* $X \subseteq toFOL(\tau')$.

To show that $\mathcal{R}_{\mathrm{FJ}}$ is complete, we must show that for a logging specification $LS = spec(X_{Guidelines}, \{\text{LoggedCall}\}) \in \textbf{Calls}$ and a source language execution τ, and a corresponding target language execution τ' that produces audit log $\mathbb{L}$, for any ground instance $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in LS(\tau)$ we have $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in \mathbb{L}$. In order to show that, we need to show that $(X \cup X_{Guidelines}) \vdash \text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$, where X is the set of logging preconditions tracked during the target language execution τ' (see Rules Precondition and Log).

A key insight is that the only facts in $toFOL(\tau)$ relevant to deriving grounded goals of the form $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$ are facts $\text{Call}(n', \mathtt{C}', \bar{\mathtt{v}}', \mathtt{m}', \mathtt{u}')$ for $\mathtt{C}'.\mathtt{m}' \in \{Logevent(LS)\} \cup Triggers(LS)$, and these are exactly the facts that appear in the instrumented program's set of logging preconditions tracked during execution. Formally, the *support of* a grounded goal ψ given assumptions X, denoted $support(X, \psi)$, is the set of conjuncts in ϕ where $\phi \Rightarrow \psi$ is a grounding of a Horn clause of the form $\forall x_1, \ldots, x_m.\ \phi' \Rightarrow \psi' \in X$ and $X \vdash \phi$. In Datalog terms, these are the grounded subgoals of ψ in its derivation given knowledge base X. Hence:

Lemma 3.2.4 *Let* $\mathtt{p}$ *be a* FJ *program and* $LS \in \textbf{Calls}$ *be a logging specification where* $LS = spec(Y, S)$*. For all* τ *such that* $\mathtt{p} \Downarrow \tau$ *there exists a target language execution trace* τ' *such that* $\tau :\approx \tau'$*,* $\mathcal{R}(\mathtt{p}, LS) \Downarrow \tau'$ *and* $\tau' = \kappa_0 \ldots \kappa_n$ *where* $\kappa_n = (\mathtt{e}, m, X, \mathbb{L})$ *such that for all* $\phi \in LS(\tau)$ *and* $\text{Call}(t, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \mathtt{u}) \in support(Y \cup toFOL(\tau), \phi)$ *we have* $\text{Call}(t, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \mathtt{u}) \in X$*.*

From Lemma 3.2.3 and Lemma 3.2.4, we can establish that the log generated by the rewritten program is the least Herbrand model of the given logging specification semantics.

Lemma 3.2.5 *Let* $\mathtt{p}$ *be a* FJ *program and* $LS = \mathit{spec}(X, \{\text{LoggedCall}\}) \in$ **Calls** *be a logging specification. For all* τ *such that* $\mathtt{p} \Downarrow \tau$ *we have* $\mathit{simlogs}(\mathcal{R}_{\text{FJ}}(\mathtt{p}, LS), \tau) = \{\mathbb{L}\}$ *such that:*

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

Proof. (Sketch.) First, note that we can construct a target language execution trace τ' such that $\mathcal{R}_{\text{FJ}}(\mathtt{p}, LS) \Downarrow \tau'$ and $\tau :\approx \tau'$ (i.e., τ' executes the source program to the same point that τ does). Let the last configuration of τ' be $(\mathtt{e}, n, Y, \mathbb{L})$. We observe that this construction uniquely defines the log $\mathbb{L}$ due to determinism in the language and Definition 4.4.8.

Let $Z = \mathfrak{H}(X \cup \mathit{toFOL}(\tau)) \cap L_{\{\text{LoggedCall}\}}$. If $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in \mathbb{L}$, then $X \cup Y \vdash \text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$ by semantic definition of FJ_{log}. But by Lemma 3.2.3 we have $X \subseteq \mathit{toFOL}(\tau)$, hence $X \cup \mathit{toFOL}(\tau) \vdash \text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}})$ so $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in Z$.

Conversely, if $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in Z$, by Lemma 3.2.4 (and the determinism of our languages), any Call fact in $\mathit{support}(X \cup \mathit{toFOL}(\tau), \text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}))$ is also in X, hence every such LoggedCall will also be in $\mathbb{L}$. Thus $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in Z$ iff $\text{LoggedCall}(n, \mathtt{C}, \bar{\mathtt{v}}, \mathtt{m}, \bar{\mathtt{u}}) \in \mathbb{L}$. The result follows by definition of $\mathfrak{H}$. $\square$

These Lemmas suffice to prove our main Theorem, demonstrating soundness and completeness of program rewriting algorithm $\mathcal{R}_{\text{FJ}}$. This result establishes that the log generated by the instrumented program and the semantics of the logging specification contain exactly the same information with respect to the sublanguage $L_{\{\text{LoggedCall}\}}$.

Theorem 3.2.3 (Soundness and Completeness) *Program rewriting algorithm* $\mathcal{R}_{\text{FJ}}$ *is sound and complete (Definitions 2.5.2 and 2.5.3).*

Proof. Let $\mathfrak{p}$ be a FJ program and $LS = spec(X, \{\text{LoggedCall}\}) \in \textbf{Calls}$ be a logging specification. We aim to show that for all source language execution traces τ such that $\mathfrak{p} \Downarrow \tau$ we have $simlogs(\mathcal{R}_{\text{FJ}}(\mathfrak{p}, LS), \tau) = \{\mathbb{L}\}$ such that $C(\mathbb{L}) = LS(\tau)$.

By Lemma 3.2.5, we have that $simlogs(\mathcal{R}_{\text{FJ}}(\mathfrak{p}, LS), \tau) = \{\mathbb{L}\}$ such that $\mathbb{L} = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{LoggedCall}\}}$. By Lemma 3.2.1 and Lemma 3.2.2 $LS(\tau) = C(C(\mathfrak{H}(X \cup toFOL(\tau))) \cap L_{\{\text{LoggedCall}\}}) = C(\mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{LoggedCall}\}})$. Hence, both $LS(\tau) \leq C(\mathbb{L})$ and $C(\mathbb{L}) \leq LS(\tau)$. □

3.3 Case Study on a Medical Records System

As a case study, we have developed a tool [81] that enables automatic instrumentation of logging specifications for the OpenMRS system. The implementation is based on the formal model developed in Section 3.2 which enjoys a correctness guarantee. The logging information is stored in a SQL database consisting of multiple tables, and the correctness of this scheme is established via the monotone mapping defined in Section 2.6.3. We have also considered how to reduce memory overhead as a central optimization challenge.

OpenMRS [15] is a Java-based open-source web application for medical records, built on the Spring Framework [82]. Previous efforts in auditing for OpenMRS include recording any modification to the database records as part of the OpenMRS core implementation, and logging every function call to a set of predefined records [72]. The latter illustrates the relevance of function invocations as a key factor in

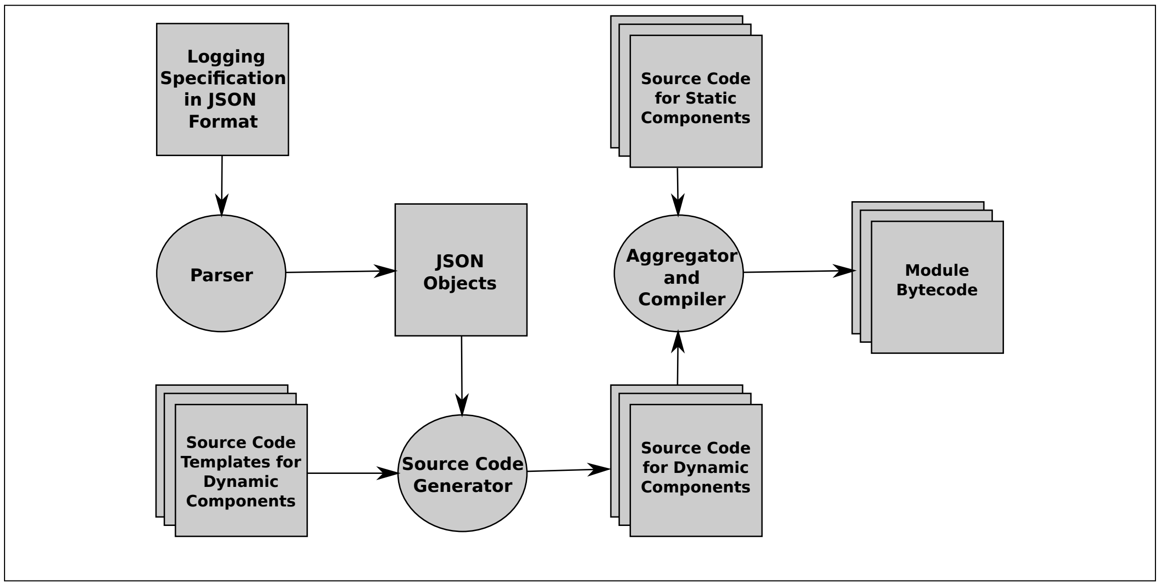
Logging Specification in JSON Format
Source Code for Static Components
Parser
JSON Objects
Aggregator and Compiler
Module Bytecode
Source Code Templates for Dynamic Components
Source Code Generator
Source Code for Dynamic Components

Figure 3.12: Module Builder

logging. Furthermore, function calls define the fundamental unit of "secure operations" in OpenMRS access control [78]. This highlights the relevance of our **Calls** logging specification class, particularly as it pertains to specification of break the glass policies, which are sensitive to authorization.

In contrast to the earlier auditing solutions for OpenMRS, ours facilitates a smart log generation mechanism in which only the necessary information are recorded, based on accurate log specifications. Moreover, logging specifications are defined independently from code, rather than being embedded in it in an ad-hoc manner. This way, system administrators need to only assert logging specifications in the style of **Calls** (Definition 3.2.1), and the tool builds the corresponding module that could be installed on the OpenMRS server. This is more convenient, declarative, and less error prone than direct ad-hoc instrumentation of code. In Figure 3.12 the details of building the module is given.

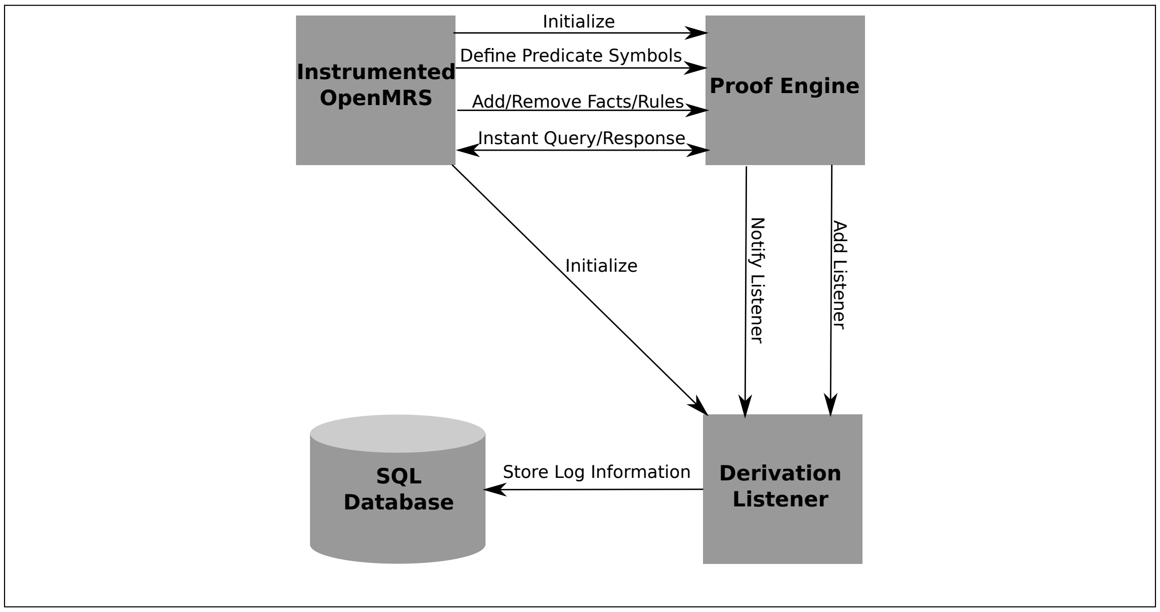

Figure 3.13: System Architecture

System Architecture Summary

To clarify the following discussion, we briefly summarize the architecture of our system. Logging specifications are made in the style of **Calls**, which can be parsed into JSON objects with a standard form recognized by our system. Instrumentation of legacy code is then accomplished using aspect oriented programming. Parsed specifications are used to identify join points, where the system weaves aspects supporting audit logging into OpenMRS bytecode. These aspects communicate with a proof engine at the joint points to reason about audit log generation, implementing the semantics developed for $\mathrm{FJ}_{\mathrm{log}}$ in Section 3.2.3. In our deployment logs are recorded in a SQL database, but our architecture supports other approaches via the use of listeners. Figure 3.13 illustrates the major components we have deployed to facilitate auditing at runtime.

3.3.1 Break the Glass Policies for OpenMRS

Break the glass policies for auditing are intended to retrospectively manage the same security that is proactively managed by access control (before the glass is broken). Thus it is important that we focus on the same resources in auditing as those focused on by access control. The data model of OpenMRS consists of several domains e.g., "Patient" and "User" domains contain information about the patients and system users respectively, and "Encounter" domain includes the information regarding the interventions of healthcare providers with patients. In order to access and modify the information in different domains, corresponding service-layer functionalities are defined that are accessible through web interface. These functionalities provide security sensitive operations through which data assets are handled. Thus, OpenMRS authorization mechanism checks user eligibility to perform these operations [78]. Likewise, we focus on these functionalities to be addressed in the logging specifications, i.e., the triggers and logging events are constrained to the service-layer methods as they provide access to data domains, e.g., the patient and user data.

We adapt the logical language of logging specifications developed above (Definition 3.2.1), with the minor extension that we allow logging of methods with more than one argument. We note that logging specifications can include other information specified as safe Horn clauses, e.g. ACLs, and generally define predicates specified in $\phi((x_0, t_0), \ldots, (x_n, t_n))$ of Definition 3.2.1. We consider break the glass policies as a key example application in our deployment. For instance a simple break the glass policy states that if the glass is broken by some low-level user, and subsequently the patient information is accessed by that user, the access should be logged. This policy

```
loggedCall(T, getPatient, U, P) :-
call(T, getPatient, U, P), call(S, breakTheGlass, U), @<(S, T), hasSecurityLevel(U, low).

hasSecurityLevel(admin, high).
hassecuritylevel(alice, low).
```

Figure 3.14: A Simple Break the Glass Policy Specified in the Proof Engine Database.

is depicted in Figure 3.14[1]. The variable `U` refers to the user, and the variable `P` refers to the patient. This specification also defines security levels for two users, `alice` and `admin`. The predicate `@<` defines the usual total ordering on integers.

To enable these policies in practice, we have added a "break the glass" button to a user menu in the OpenMRS GUI that can be manually activated on demand. Activation invokes the `breakTheGlass` method parameterized by the user id. We note that breaking the glass does not turn off access control in our current implementation, which we consider a separate engineering concern that is out of scope for this work.

It is worth mentioning that while our tool is designed for OpenMRS, our general approach can be used for arbitrary Java code at source or bytecode level.

3.3.2 Code Instrumentation

To instrument code for log generation, we leverage the Spring Framework that supports aspect-oriented programming (AOP). AOP is used to rewrite code where necessary with "advice", which in our case is *before* certain method invocations (so-called "before advice"). Our advice checks the invoked method names and implements the semantics given in Section 3.2.3, establishing correctness of audit logging. Join points

[1]We use the monospace font to present policies, data, and code excerpts that are relevant to the implementation.

```
<advice>
<point>org.openmrs.api.PatientService</point>
<class>
org.openmrs.module.retrosecurity.advice.RetroSecurityAdvice
</class>
</advice>
```

Figure 3.15: Specifying Joint Points for Advices.

are automatically extracted from logging specifications, and defined with service-level granularity in a configuration file. Weaving into bytecode is also performed automatically by our system.

Since the generated code pieces are before advices, they are interposed before every interface method of the declared services. An aspect is configured by declaring where the join point and corresponding advice is. For example, Figure 3.15 shows an excerpt of a configuration file, where every interface method of the service `PatientService` is a join point so before invoking each of those methods the advice in `RetroSecurityAdvice` will be woven into the control flow.

The Advice

`RetroSecurityAdvice` is the before advice automatically generated by our system based on the logging specification. It essentially determines whether a method call is a trigger or a logging event and interacts with the proof engine appropriately in each case.

The first time the advice is executed, the proof engine is initialized in a separate thread. Moreover, a LoggedCall derivation listener is added to the list of the engine listeners. Then, if memory overhead mitigation (Section 3.4) is not activated, the invoked method names are checked and the rule Protection (Section 3.2.3) is implemented for the triggers and the logging event, i.e., the proof engine is asked to add the the information regarding the invocation of the method. In the case memory overhead mitigation is activated, the set of Protection rules of Figure 3.17 are implemented for the triggers and the logging event. The implementation of the rules Log and NoLog (Section 3.2.3) is handled by the LoggedCall derivation listener.

The advice also checks for the invocation of the interface method `queryLog()`. This method communicates with the engine to facilitate instant querying based on the invocations of the logging preconditions that exist in the memory.

3.3.3 Proof Engine

According to the the semantics of $\mathrm{FJ}_{\mathrm{log}}$, it is necessary to perform logical deduction, in particular resolution of LoggedCall predicates. As we will show in Section 3.4, the required deductions could be generalized to any arbitrary formula. To this end, we have employed XSB Prolog [83] as our proof engine, due to its reliability and robustness. We have restricted our specifications to safe Horn clauses though, despite the fact that XSB Prolog provides a more expressive tool. In order to have a bidirectional communication between the Java application and the engine, InterProlog Java/Prolog SDK [84] is used. A lightweight Datalog engine that communicates with Java application is more preferable, but we were not able to identify such third-party tool and thus it remains as future work.

The proof engine is initialized in a separate thread with an interface to the main execution trace. The interface includes methods to define predicates, to add rules and facts, and to revoke them asynchronously[2]. The asynchrony avoids blocking the "normal" execution trace for audit logging purposes. The interface also provides an

[2]Revoking facts is required for memory overhead mitigation.

instant querying mechanism. The execution trace of instrumented program communicates with the XSB Prolog engine as these interface methods are invoked in the advices.

3.3.4 Writing and Storing the Log

Asynchronous communication with the proof engine through multi-threading enables us to modularize the deduction of the information that we need to log, separate from the storage and retainment details. This supports a variety of possible approaches to storing log information– e.g., using a strict transactional discipline to ensure writing to critical log, and/or blocking execution until log write occurs. Advice generated by the system for audit log generation just needs to include event listeners to implement the technology of choice for log storage and retainment.

In our application, the logging information is stored in a SQL database consisting of multiple tables. The generated advices include event listeners to implement our technology of choice for log storage and retainment. In case a new logging information is derived by the proof engine, the corresponding listeners in the main execution trace are notified and the listeners partition and store the logging information in potentially multiple tables. Correctness of this storage technique is established using the monotone mapping *rel* defined in Section 2.6.3, i.e., the join of these tables are information equivalent (Definition 2.1.4) to the semantics of logging specification for a given break the glass policy. This ensures that the correctness guarantees extend to database storage.

Consider the case where a `loggedCall` is derived by the proof engine given the logging specification in Section 3.3.1. Here, the instantiation of `U` and `P` are user

```
select time, "getPatien", uname, patient_name
from GetPatL, User, Patient
where GetPatL.uid = User.uid and GetPatL.pid = Patient.pid
```

Figure 3.16: Querying the Log with SQL.

and patient names, respectively, used in the OpenMRS implementation. However, logged calls are stored in a table called `GetPatL` with attributes `time`, `uid`, and `pid`, where `uid` is the primary key for a `User` table with a `uname` attribute, and `pid` is the primary key for a `Patient` table with a `patient_name` attribute. Thus, for any given logging specification of the appropriate form, the monotonic mapping *rel* of the `select` statement in Figure 3.16 gives us the exact information content of the logging specification following execution of an OpenMRS session:

3.4 Reducing Memory Overhead

A source of overhead in our system is memory needed to store logging preconditions. We observe that a naive implementation of the intended semantics will add all trigger functions to the logging preconditions, regardless of whether they are redundant in some way. To optimize memory usage, we therefore aim to refrain from adding information about trigger invocations if it is unnecessary for future derivations of audit log information. As a simple example, in the following logging specification it suffices to add only the first invocation of $\mathtt{C}_1.\mathtt{m}_1$ to the set of logging preconditions to infer the relevant logging information.

$$\forall t_0, t_1, \bar{\mathtt{x}}_0, \bar{\mathtt{x}}_1, \bar{\mathtt{y}}_0, \bar{\mathtt{y}}_1 \,.\, \mathrm{Call}(t_0, \mathtt{C}_0, \bar{\mathtt{x}}_0, \mathtt{m}_0, \bar{\mathtt{y}}_0) \wedge \mathrm{Call}(t_1, \mathtt{C}_1, (t_1, \mathtt{C}_1, \bar{\mathtt{x}}_1, \mathtt{m}_1, \bar{\mathtt{y}}_1)) \wedge t_1 < t_0$$
$$\implies \mathrm{LoggedCall}(t_0, \mathtt{C}_0, \bar{\mathtt{x}}_0, \mathtt{m}_0, \bar{\mathtt{y}}_0).$$

Intuitively, our general approach is to rewrite the body of a given logging specification in a form consisting of different conjuncts, such that the truth valuation of each conjunct is independent of the others. This way, the required information to derive each conjunct is independent of the information required for other conjuncts. Then, if the inference of a LoggedCall predicate needs a conjunct to be derived only once during the program execution, we can limit the amount of information required to derive that conjunct to the point where it is derivable, without affecting the derivability of other conjuncts. In other words, following derivation of that conjunct, triggers in the conjunct are "turned off", i.e. no longer added to logging preconditions when encountered during execution.

Formally, the logging specification is rewritten in the form

$$\forall t_0, \ldots, t_n, \bar{\mathtt{x}}_0, \ldots, \bar{\mathtt{x}}_n, \bar{\mathtt{y}}_0, \ldots, \bar{\mathtt{y}}_n \,.\, \bigwedge_{i=1}^{n} (t_i < t_0) \bigwedge_{k=1}^{L} Q_k \implies \mathrm{LoggedCall}(t_0, \mathtt{C}_0, \bar{\mathtt{x}}_0, \mathtt{m}_0, \bar{\mathtt{y}}_0),$$

where each Q_k is a conjunct of literals with independent truth valuation resting on disjointness of predicated variables. In what follows, a formal description of the technique is given.

Since we have a linear computational model, the predicates corresponding to the timestamp comparisons $(t_i < t_0)$ do not play a significant role in the inference of LoggedCall predicates. There reason is, at any point in time, the set of logging preconditions only contain function invocations that have been occurred in the past, i.e., if the logging event is invoked at timestamp t_0, then $t_i < t_0$ holds for all trigger invocation timetamps t_i that are retained in the set of logging preconditions. In what follows, a formal description of the technique is given.

Consider the Definition 3.2.1, where $\mathtt{C}_1.\mathtt{m}_1 = \mathit{Logevent}(\mathit{LS})$. Let predicates of the form $\varphi_{i'}((\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0), \ldots, (\bar{\mathtt{x}}_n, \bar{\mathtt{y}}_n, t_n))$ be positive literals and

$$\phi((\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0), \ldots, (\bar{\mathtt{x}}_n, \bar{\mathtt{y}}_0, t_n)) \triangleq \bigwedge_{i'=1}^{n'} \varphi_{i'}((\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0), \ldots, (\bar{\mathtt{x}}_n, \bar{\mathtt{y}}_n, t_n)).$$

Then, we define the set Ψ as

$$\begin{aligned}\Psi \triangleq &\{\varphi_{i'}((\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0), \ldots, (\bar{\mathtt{x}}_n, \bar{\mathtt{y}}_n, t_n)) \mid i' \in 1 \cdots n'\} \cup \\ &\{(\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)) \mid i \in 0 \cdots n\}.\end{aligned}$$

Moreover, let's denote the set of free variables of a formula ϕ as $FV(\phi)$, and abuse this notation to represent the set of free variables that exist in a set of formulas. Then, $FV(\Psi) = \{\bar{\mathtt{x}}_0, \cdots, \bar{\mathtt{x}}_n, \bar{\mathtt{y}}_0, \cdots, \bar{\mathtt{y}}_n, t_0, \cdots, t_n\}$. Next, we define the relation, $\circledast_{FV}$ over free variables of positive literals in Ψ, which represents whether they are free variables of the same literal.

Definition 3.4.1 *Let $\circledast_{FV} \subseteq FV(\Psi) \times FV(\Psi)$ be a relation where $\alpha \circledast_{FV} \beta$ iff there exists some literal $P \in \Psi$ such that $\alpha, \beta \in FV(P)$. Then, the transitive closure of $\circledast_{FV}$ is denoted by $\circledast_{TFV}$.*

Lemma 3.4.1 *$\circledast_{FV}$ is reflexive and symmetric.*

Corollary 3.4.1 *$\circledast_{TFV}$ is an equivalence relation and so, partitions $FV(\Psi)$*

Let $[\alpha]_{\circledast_{TFV}}$ denote the equivalence class induced by $\circledast_{TFV}$ over $FV(\Psi)$, where $[\alpha]_{\circledast_{TFV}} \triangleq \{\beta \mid \alpha \circledast_{TFV} \beta\}$. Intuitively, each equivalence class $[\alpha]_{\circledast_{TFV}}$ represents a set of free variables in Ψ that are free in a subset of literals of Ψ, transitively. To be explicit about these subsets of literals, we have the following definition (Definition

3.4.2). Note that rather than representing an equivalence class using a representative α (i.e., the notation $[\alpha]_{\circledast_{TFV}}$), we may employ an enumeration of these classes and denote each class as C_k, where $k \in 1 \cdots L$. L represents the number of equivalence classes that have partitioned $FV(\Psi)$. In order to map these two notations, we consider a mapping $\omega : FV(\Psi) \to \{1, \cdots, L\}$ where $\omega(\alpha) = k$ if $[\alpha]_{\circledast_{TFV}} = C_k$.

Definition 3.4.2 *Let C be an equivalence class induced by $\circledast_{TFV}$. The predicate class $\mathcal{P}_C$ is a subset of literals of Ψ defined as $\mathcal{P}_C \triangleq \{P \in \Psi \mid FV(P) \subseteq C\}$. We define the independent conjuncts as $Q_C \triangleq \bigwedge_{P \in \mathcal{P}_C} P$. We also denote $Q_{[\alpha]}$ as Q_k if $\omega(\alpha) = k$. Obviously, $FV(Q_k) = C_k$.*

Lemma 3.4.2 *Let $C_1, \cdots, C_L$ be all the equivalence classes induced by $\circledast_{TFV}$ over $FV(\Psi)$. Then, $\mathcal{P}_{C_1}, \cdots, \mathcal{P}_{C_L}$ give a partition on Ψ.*

Proof. We need to show that

- for all distinct $k, k' \in 1 \cdots L$, $\mathcal{P}_{C_k} \cap \mathcal{P}_{C_{k'}} = \emptyset$:

  By contradiction: Let $k, k' \in 1 \cdots L$ be specific distinct indexes where $\mathcal{P}_{C_k} \cap \mathcal{P}_{C_{k'}} \neq \emptyset$, i.e., there exists some $P \in \Psi$, such that $P \in \mathcal{P}_{C_k}$ and $P \in \mathcal{P}_{C_{k'}}$. Then, according to the definition, we have $FV(P) \subseteq C_k$ and $FV(P) \subseteq C_{k'}$. Since $FV(P)$ is non-empty, we would have $C_k \cap C_{k'} \neq \emptyset$, which contradicts with C_k and $C_{k'}$ being classes over $FV(\Psi)$.

- $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} = \Psi$:

  Obviously, $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} \subseteq \Psi$by the definition of predicate classes. It only suffices to show that $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} \supseteq \Psi$. Let $P \in \Psi$. Since $FV(P) \neq \emptyset$, there exists some $\alpha \in FV(P)$. Considering the equivalence class $[\alpha]_{\circledast_{TFV}}$, we will then have $FV(P) \subseteq [\alpha]_{\circledast_{TFV}}$. This entails that $P \in \mathcal{P}_{[\alpha]_{\circledast_{TFV}}}$ and so, $P \in \bigcup_{k=1}^{L} \mathcal{P}_{C_k}$.

□

In order to specify and prove the correctness of the proposed technique, a new calculus $\mathrm{FJ}'_{\mathrm{log}}$ is formalized with memory overhead mitigation capabilities. In what follows the details of this calculus and the correctness result are given. Moreover, a developed example of how these techniques could be applied to a sample logging specification in **Calls** is discussed, later in this section.

The given techniques are implemented in the OpenMRS retrospective security module as a case study.

3.4.1 Language with Memory Overhead Mitigation

The language $\mathrm{FJ}_{\mathrm{log}}$ is defined in Section 3.2 whose syntax includes a command to track logging preconditions ($\mathtt{callEvent}(\mathtt{m}, \bar{\mathtt{u}})$) and a command to emit log entries ($\mathtt{emit}(\mathtt{m}, \bar{\mathtt{u}})$). Configurations are quadruples of the form $\kappa ::= (\mathtt{e}, n, X, \mathbb{L})$ which include a set X of logging precondtions (sometimes referred to as "database"), and an audit log $\mathbb{L}$. The semantics of $\mathrm{FJ}_{\mathrm{log}}$ includes the rule Precondition to update the set of logging preconditions.

The language $\mathrm{FJ}'_{\mathrm{log}}$ has the same syntax as $\mathrm{FJ}_{\mathrm{log}}$. The configurations, however, have an additional component W which is a set of function names. It is used to keep track of functions that we do not require to add their invocation information to the database any more. By adding some trigger name $\mathtt{C}_i.\mathtt{m}_i$ to W, we indicate that further additions of information regarding $\mathtt{C}_i.\mathtt{m}_i$ invocations to X will not cause new

LoggedCall predicates to be derived.

$$\kappa ::= (\mathtt{e}, n, X, \mathbb{L}, W) \qquad\qquad \textit{configurations}$$

All stepwise reduction rules in this language are the same as the ones in $\mathrm{FJ_{log}}$, except for Precondition. Instead of that rule, we impose the set of rules in Figure 3.17. Note that X_G denotes the guidelines database. For the sake of brevity, we will omit $\circledast_{TFV}$ from the class notations onward.

The rule Precondition-1 states that if a trigger is invoked, but is already added to the set W, according to the semantics of W, we do not add the invocation to the database. The remaining rules consider the other case, i.e., the trigger is not already added to W. The rule Precondition-2 expresses the case where the trigger $\mathtt{C}_i.\mathtt{m}_i$ is not in W, and there are no literals except for $\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)$ with $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ or t_i as free variables. In this case, the invocation is added to database and the trigger name is added to the set W, in order to avoid further addition of invocations to this trigger. If there are literals other than $\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)$ with free variables $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ or t_i, but the free variables of all those literals are restricted to $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i, we study the derivability of the ground form of $Q_{[t_i]}$ considering the new invocation. Notice that $FV(Q_{[t_i]}) = \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\}$. If the ground form of $Q_{[t_i]}$ is derivable, then the invocation is added to the database. The trigger name is also added to W (Precondition-4). Otherwise, the invocation is not added to the database (Precondition-3). The reason is, keeping the invocation information in the database will not help deriving a ground form of $Q_{[t_i]}$ in the future steps.

If there are literals other than $\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)$ with free variables $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ or t_i, and the free variables of those literals are not restricted to $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i, but exclude $\bar{\mathtt{x}}_0$,

$\bar{\mathtt{y}}_0$ and t_0, then the derivability of $Q_{[t_i]}$ is studied. In this case, the set of free variables of $Q_{[t_i]}$ is $[t_i]$, for sure. If a ground form of $Q_{[t_i]}$ is derivable, then the invocation is added to the database and the trigger names whose timestamp and argument variable are in $[t_i]$ are added to W (Precondition-6). Otherwise, the invocation is still added to the database (Precondition-5) but the trigger name is not added to W. The reason is, keeping the invocation information in the database might help derive a ground form of $Q_{[t_i]}$ in the future steps, since there exist free variables other than $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i in $Q_{[t_i]}$ that could be substituted with proper values so that $Q_{[t_i]}$ could be derived. Note that $\bar{\alpha}_i$ represents a sequence of free variables $[t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\}$ and $\bar{a}_i$ is a sequence of timestamps and values, except for the timestamp and argument value of trigger i. Moreover, $\left[\bar{a}_i/\bar{\alpha}_i\right]$ denotes the substitution of values to their corresponding variables.

The rule Precondition-7 discusses the remaining case for triggers, that is when there are literals other than $\text{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)$ with free variables $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ or t_i, the free variables of those literals are not restricted to $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i, and include $\bar{\mathtt{x}}_0$, $\bar{\mathtt{y}}_0$ or t_0. Then, the invocation is added to the database but the trigger name is not added to W, independent of whether a ground form of $Q_{[t_i]}$ is derivable or not. If a ground form of $Q_{[t_i]}$ is not derivable at the moment, keeping the invocation information in the database might help derive a ground form of $Q_{[t_i]}$ in the future steps, since there exist free variables other than $\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i in $Q_{[t_i]}$ that could be substituted with proper values so that a ground form of $Q_{[t_i]}$ could be derived. Otherwise, if a ground form of $Q_{[t_i]}$ is derivable, we might still need to add future invocations of $\mathtt{C}_i.\mathtt{m}_i$ and other triggers whose timestamp and argument variables are in $[t_i]$. That is why, we avoid adding trigger names to W. This is due to the fact that $Q_{[t_i]}$ includes invocation to the logging event and possibly other predicates defined over its timestamp and argument variable ($\bar{\mathtt{x}}_i$, $\bar{\mathtt{y}}_i$ and t_i). Thus, future derivations of $Q_{[t_i]}$ could be affected.

This represents a major difference between the case when $Q_{[t_i]}$ includes $\{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\}$ and the case $Q_{[t_i]}$ excludes these variables. In the latter case, it is only required to derive a ground form of $Q_{[t_i]}$ once during program execution, in order to study whether LoggedCall predicates could be inferred. Therefore, whenever a ground form of $Q_{[t_i]}$ is derivable at the time of $\mathtt{C}_i.\mathtt{m}_i$ invocation, W is beefed up with the corresponding trigger names. In the prior case, however, it is required to derive all possible ground forms of $Q_{[t_i]}$, so that we would be able to infer all possible LoggedCall predicates.

The last rule (Precondition-8) discusses the case where the logging event is invoked. Since we need to infer all possible LoggedCall predicates, we add all those invocations to the database.

3.4.2 Correctness of Memory Overhead Mitigation

In order to study the executional behaviour of programs in $\mathrm{FJ}'_{\mathrm{log}}$ compared to the case where they are executing in $\mathrm{FJ}_{\mathrm{log}}$, we need to understand the relationship between the set of logging preconditions in these languages. To this end, we develop an algorithm that generates the reduced database and the set W of trigger names, out of a full-blown database of logging preconditions. The algorithm Refine is defined in Algorithm 2. We denote the reduced set of logging preconditions X as $\mathcal{R}(X)$, and the generated set of trigger names as $\mathcal{W}(X)$, defined as follows:

$$\mathcal{R}(X) \triangleq \mathrm{fst}(\mathrm{Refine}\ X\ [\,]\ \emptyset) \qquad\qquad \mathcal{W}(X) \triangleq \mathrm{snd}(\mathrm{Refine}\ X\ [\,]\ \emptyset)$$

We do not express any explicit mapping between sets and sorted lists in our formulation for the sake of brevity. The employment of sets and their corresponding sorted lists are clear from the context. Let's denote the restriction of a set X to timestamps less than or equal to n, as $X|_n$.

In what follows, Lemmas 3.4.3 to 3.4.7 discuss properties of $\mathcal{R}(X)$ and $\mathcal{W}(X)$. Lemma 3.4.7, in particular, shows that $\mathcal{R}(X)$ is enough to derive all LoggedCall predicates that are derivable from X. Lemma 3.4.8 states that in a single reduction step, the reduced set of logging preconditions is preserved and the generated audit log is maintained, which then can be generalized straightforwardly to multi-step reduction, in Theorem 3.4.1. Then, Corollary 3.4.2 gives us our intended result, which states that a program could be evaluated in $\mathrm{FJ}'_{\mathrm{log}}$ with reduced set of logging preconditions and the same audit log.

Lemma 3.4.3 *Let X be a set of logging preconditions. For all $i \in 1 \cdots n$, if $t_i \in [t_0]$ then $\mathtt{C}_i.\mathtt{m}_i \notin \mathcal{W}(X)$.*

Proof. Since $t_i \in [t_0]$, $[t_i] = [t_0]$ and so $[t_i] \cap \{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\} \neq \emptyset$. Thus, for each invocation information of $\mathtt{C}_i.\mathtt{m}_i$ in X, only line 25 of Algorithm 2 is executed. Obviously, $\mathtt{C}_i.\mathtt{m}_i$ is not added to W in this line. □

Lemma 3.4.4 *For all $i \in 0 \cdots n$, if $t_i \in [t_0]$ and* $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in X$*, then* $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$*.*

Proof. First let's consider the case where $i = 0$. Then, according to line 31 of Algorithm 2, $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$. Now, let $i \in 1 \cdots n$. According to Lemma 3.4.3, $\mathtt{C}_i.\mathtt{m}_i \notin \mathcal{W}(X)$ and so $\mathtt{C}_i.\mathtt{m}_i \notin \mathcal{W}(X|_{n-1})$. This implies that only line 25 of Algorithm

Algorithm 2: Refine algorithm

```
Input: Sorted list of invocation facts, Sorted list of invocation facts, Set of
       trigger names
Output: Sorted list of invocation facts, Set of trigger names
Refine [ ] $Y$ $W$ = $(Y, W)$

Refine $((\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)) :: X)$ $Y$ $W$ =
if $i \in 1 \cdots n$ then
    if $\mathtt{C}_i.\mathtt{m}_i \in W$ then
        Refine $X$ $Y$ $W$
    else
        if $\mathcal{P}_{[t_i]} - \{\text{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} = \emptyset$ then
            Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $(W \cup \{\mathtt{C}_i.\mathtt{m}_i\})$
        else
            if $[t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} = \emptyset$ then
                if
                $Y \cup \{\text{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\} \cup X_G \nvdash Q_{[t_i]}\Big[n-1/t_i\Big]\Big[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\Big]\Big[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\Big]$
                then
                    Refine $X$ $Y$ $W$
                else
                    Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $(W \cup \{\mathtt{C}_i.\mathtt{m}_i\})$
                end
            else
                if $[t_i] \cap \{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\} = \emptyset$ then
                    if $\nexists \bar{a}_i \,.\, Y \cup \{\text{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\} \cup X_G \vdash$
                    $Q_{[t_i]}\Big[\bar{a}_i/\bar{\alpha}_i\Big]\Big[n-1/t_i\Big]\Big[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\Big]\Big[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\Big]$ then
                        Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $W$
                    else
                        Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $(W \cup_{t_{i'} \in [t_i]} \{\mathtt{C}_{i'}.\mathtt{m}_{i'}\})$
                    end
                else
                    Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $W$
                end
            end
        end
    end
else
    Refine $X$ $(Y + [\text{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)])$ $W$
end
```

2 can be executed for $\mathtt{C}_i.\mathtt{m}_i$, in which the invocation to $\mathtt{C}_i.\mathtt{m}_i$ is added to Y, which then is reflected in $\mathcal{R}(X)$. $\square$

Let $\bar{\bar{t}}(k)$, $\bar{\bar{\mathtt{x}}}(k)$ and $\bar{\bar{\mathtt{y}}}(k)$ denote the sequences of timestamp, object field variables and function argument variables respectively, that are in class C_k. Similarly, $\bar{\bar{s}}(k)$ and $\bar{\bar{\mathtt{v}}}(k)$ and $\bar{\bar{\mathtt{u}}}(k)$ are used to denote sequences of timestamps, field values and function argument values that substitute $\bar{\bar{t}}(k)$, $\bar{\bar{\mathtt{x}}}(k)$ and $\bar{\bar{\mathtt{y}}}(k)$ respectively.

Lemma 3.4.5 *Let $i \in 1 \cdots n$. Suppose that*

$$X \cup X_G \vdash Q_{\omega(i)}\Big[\bar{\bar{s}}(\omega(i))/\bar{\bar{t}}(\omega(i))\Big]\Big[\bar{\bar{\mathtt{v}}}(\omega(i))/\bar{\bar{\mathtt{x}}}(\omega(i))\Big]\Big[\bar{\bar{\mathtt{u}}}(\omega(i))/\bar{\bar{\mathtt{y}}}(\omega(i))\Big],$$

for some sequences of timestamps, field values and argument values $\bar{\bar{s}}(\omega(i))$, $\bar{\bar{\mathtt{v}}}(\omega(i))$ and $\bar{\bar{\mathtt{u}}}(\omega(i))$. If $\mathrm{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in X$ and $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \notin \mathcal{R}(X)$ for some i' such that $t_{i'} \in [t_i]$, then $\mathtt{C}_{i'}.\mathtt{m}_{i'} \in \mathcal{W}(X|_{s_{i'}-1})$.

Proof. Let's assume $\mathtt{C}_{i'}.\mathtt{m}_{i'} \notin \mathcal{W}(X|_{s_{i'}-1})$. Since $\mathrm{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in X$ and $\mathrm{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \notin \mathcal{R}(X)$, we need to follow Refine algorithm to extract the places where the invocation information is not added to Y. The only place with such a property is line 13 (other than line 6 which is refuted by the assumption). Then,

- $\mathcal{P}_{[t_{i'}]} - \{\mathrm{Call}(t_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{x}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{y}}_{i'})\} = \emptyset$, $[t_{i'}] - \{\bar{\mathtt{x}}_{i'}, \bar{\mathtt{y}}_{i'}, t_{i'}\} = \emptyset$, and
- $X|_{s_{i'}-1} \cup \{\mathrm{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'})\} \cup X_G \nvdash Q_{[t_{i'}]}\Big[s_{i'} - 1/t_{i'}\Big]\Big[\bar{\mathtt{v}}_{i'}/\bar{\mathtt{x}}_{i'}\Big]\Big[\bar{\mathtt{u}}_{i'}/\bar{\mathtt{y}}_{i'}\Big]$.

The latter result is in contradiction with the general form

$$X \cup X_G \vdash Q_{\omega(i)}\Big[\bar{\bar{s}}(\omega(i))/\bar{\bar{t}}(\omega(i))\Big]\Big[\bar{\bar{\mathtt{v}}}(\omega(i))/\bar{\bar{\mathtt{x}}}(\omega(i))\Big]\Big[\bar{\bar{\mathtt{u}}}(\omega(i))/\bar{\bar{\mathtt{y}}}(\omega(i))\Big]$$

considering the fact that $\omega(i) = \omega(i')$ due to $t_{i'} \in [t_i]$, and also $Q_{[t_i]} = Q_{\omega(i')}$. □

Lemma 3.4.6 *Let X be a set of logging preconditions. For all $i \in 1 \cdots n$, if $\mathtt{C}_i.\mathtt{m}_i \in \mathcal{W}(X)$, then there exist some n, $\bar{\mathtt{v}}_i$, $\bar{\mathtt{u}}_i$ and $\bar{a}_i$ such that* $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$ *and*

$$\mathcal{R}(X) \cup X_G \vdash Q_{[t_i]}\Big[\bar{a}_i/\bar{\alpha}_i\Big]\Big[n/t_i\Big]\Big[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\Big]\Big[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\Big].$$

Proof. The only places where a trigger is added to W in Algorithm 2 are the lines 9, 15, and 22. In line 9, invocation to $\mathtt{C}_i.\mathtt{m}_i$ is added to Y, which then is reflected in $\mathcal{R}(X)$. Moreover, this line is executed whenever $\mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} = \emptyset$. Thus, for this case $Q_{[t_i]} = \mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)$. Then, since $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$, we have $\mathcal{R}(X) \cup X_G \vdash \mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)$. In line 15, similar to line 9, invocation to $\mathbf{g}_i$ is added to Y, so $\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$. Moreover, this line is executed provided the condition in lines 12 does not hold. This ensures the derivation of the ground form of $Q_{[t_i]}$. The line 22 is executed if the condition in lines 19 does not hold. Therefore, a ground form of $Q_{[t_i]}$ should be derivable. This entails that for all $t_{i'} \in [t_i]$, there exist some n', $\bar{\mathtt{v}}_{i'}$ and $\bar{\mathtt{u}}_{i'}$ such that $\mathrm{Call}(n', \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in Y \cup \{\mathrm{Call}(n, \mathbf{g}_i, v)\}$. As $Y \cup \{\mathrm{Call}(n, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}$ is reflected in $\mathcal{R}(X)$, the proof is complete. □

Lemma 3.4.7 *If* $X \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0)\}$ *then*

$$\mathcal{R}(X) \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0).$$

Proof. If

$$X \cup X_G \vdash \text{LoggedCall}(s_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0)$$

then there exist $s_1, \cdots, s_n, \bar{\mathtt{v}}_1, \cdots, \bar{\mathtt{v}}_n, \bar{\mathtt{u}}_1, \cdots, \bar{\mathtt{u}}_n$ such that

$$X \cup X_G \vdash \bigwedge_{i=1}^{n} (s_i < s_0) \bigwedge_{k=1}^{L} Q_k \Big[\bar{\bar{s}}(k)/\bar{\bar{t}}(k)\Big] \Big[\bar{\bar{\mathtt{v}}}(k)/\bar{\bar{\mathtt{x}}}(k)\Big] \Big[\bar{\bar{\mathtt{u}}}(k)/\bar{\bar{\mathtt{y}}}(k)\Big].$$

It implies that for all $i \in 0 \cdots n$, $\text{Call}(s_i, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in X$. For each $i \in 0 \cdots n$, we consider the following two cases:

- $[t_i] = [t_0]$: Then, $t_i \in [t_0]$. Since $\text{Call}(s_i, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in X$, Lemma 3.4.4 implies that $\text{Call}(s_i, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i) \in \mathcal{R}(X)$. Then,

$$\mathcal{R}(X) \cup X_G \vdash \\ (s_i < s_0) \wedge Q_{\omega(i)} \Big[\bar{\bar{s}}(\omega(i))/\bar{\bar{t}}(\omega(i))\Big] \Big[\bar{\bar{\mathtt{v}}}(\omega(i))/\bar{\bar{\mathtt{x}}}(\omega(i))\Big] \Big[\bar{\bar{\mathtt{u}}}(\omega(i))/\bar{\bar{\mathtt{y}}}(\omega(i))\Big].$$

- $[t_i] \neq [t_0]$: Then, $[t_i] \cap \{x_0, t_0\} = \emptyset$. Now, we consider the following two subcases:
  - For all $t_{i'} \in [t_i]$, if $\text{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in X$ then

$$\text{Call}(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in \mathcal{R}(X).$$

    Then obviously the following are provable under the assumption $\mathcal{R}(X) \cup X_G$.:
    * $(s_i < s_0)$, and
    * $Q_{\omega(i)} \Big[\bar{\bar{s}}(\omega(i))/\bar{\bar{t}}(\omega(i))\Big] \Big[\bar{\bar{\mathtt{v}}}(\omega(i))/\bar{\bar{\mathtt{x}}}(\omega(i))\Big] \Big[\bar{\bar{\mathtt{u}}}(\omega(i))/\bar{\bar{\mathtt{y}}}(\omega(i))\Big]$.

- There exists some $t_{i'} \in [t_i]$ such that
    - Call$(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in X$ and
    - Call$(s_{i'}, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \notin \mathcal{R}(X)$.

  Then, by Lemma 3.4.5, $\mathtt{C}_{i'}.\mathtt{m}_{i'} \in \mathcal{W}(X|_{s_{i'}-1})$, which implies that $\mathtt{C}_{i'}.\mathtt{m}_{i'} \in \mathcal{W}(X)$. Using Lemma 3.4.6, we conclude that there exist some $n \leq s_{i'} - 1$, $\bar{\mathtt{v}}_{i'}$, $\bar{\mathtt{u}}_{i'}$ and $\bar{a}_{i'}$ such that Call$(n, \mathtt{C}_{i'}, \bar{\mathtt{v}}_{i'}, \mathtt{m}_{i'}, \bar{\mathtt{u}}_{i'}) \in \mathcal{R}(X)$ and $\mathcal{R}(X) \cup X_G \vdash Q_{[t_{i'}]}\Big[\bar{a}_{i'}/\bar{\alpha}_{i'}\Big]\Big[n/t_{i'}\Big]\Big[\bar{\mathtt{v}}_{i'}/\bar{\mathtt{x}}_{i'}\Big]\Big[\bar{\mathtt{u}}_{i'}/\bar{\mathtt{y}}_{i'}\Big]$. This entails that $\mathcal{R}(X) \cup X_G \vdash (n < s_0) \wedge Q_{\omega(i)}\Big[\bar{a}_{i'}/\bar{\alpha}_{i'}\Big]\Big[n/t_{i'}\Big]\Big[\bar{\mathtt{v}}_{i'}/\bar{\mathtt{x}}_{i'}\Big]\Big[\bar{\mathtt{u}}_{i'}/\bar{\mathtt{y}}_{i'}\Big]$ considering the fact that $Q_{\omega(i)} = Q_{[t_{i'}]}$, as $t_{i'} \in [t_i]$.

Thus, $\mathcal{R}(X) \cup X_G$ suffices to derive ground forms of all $Q_{\omega(i)}$ and therefore $\mathcal{R}(X) \cup X_G \vdash$ LoggedCall$(s_0, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0)$. □

Lemma 3.4.8 *If* $(\mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}', n', X', \mathbb{L}')$ *in* FJ$_{\text{log}}$*, then*

$$(\mathtt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)) \to (\mathtt{e}', n', \mathcal{R}(X'), \mathbb{L}', \mathcal{W}(X'))$$

in FJ$'_{\text{log}}$.

Proof. By induction on the derivation of $(\mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}', n', X', \mathbb{L}')$. The interesting cases are the reductions of $\mathtt{new}\ \mathtt{C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathit{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i)$ and $\mathtt{new}\ \mathtt{C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathit{emit}(\mathtt{m}_i, \bar{\mathtt{u}}_i)$.

Let

$$(\mathtt{new}\ \mathtt{C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathit{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}) \to (\mathtt{e}, n, X \cup \{\text{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, \mathbb{L}).$$

There are eight cases in FJ$'_{\text{log}}$, then. The first case is where $i \in 1 \cdots n$ and $\mathtt{C}_i.\mathtt{m}_i \in$

$\mathcal{W}(X)$. Then,

$$(\texttt{new C}_\texttt{i}(\bar{\texttt{v}}_\texttt{i}).\mathit{callEvent}(\texttt{m}_i, \bar{\texttt{u}}_i); \texttt{e}, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \to (\texttt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)).$$

Then we need to show that $\mathcal{R}(X) = \mathcal{R}(X \cup \{\text{Call}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)\})$ and $\mathcal{W}(X) = \mathcal{W}(X \cup \{\text{Call}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)\})$ for this case, which hold based on line 6 of Refine in Algorithm 2. The other seven cases are similarly provable based on the definition of Refine.

Let

$$(\texttt{new C}_\texttt{i}(\bar{\texttt{v}}_\texttt{i}).\mathit{emit}(\texttt{m}_i, \bar{\texttt{u}}_i); \texttt{e}, n, X, \mathbb{L}) \to (\texttt{e}, n, X, \mathbb{L} \cup \{\text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)\}).$$

This holds when $X \cup X_G \vdash \text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)$. Using Lemma 3.4.7, we then have $\mathcal{R}(X) \cup X_G \vdash \text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)$, which implies that

$$\begin{aligned}(\texttt{new C}_\texttt{i}(\bar{\texttt{v}}_\texttt{i}).\mathit{emit}(\texttt{m}_i, \bar{\texttt{u}}_i); \texttt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)) \to \\ (\texttt{e}, n, \mathcal{R}(X), \mathbb{L} \cup \{\text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)\}, \mathcal{W}(X)).\end{aligned}$$

Let

$$(\texttt{new C}_\texttt{i}(\bar{\texttt{v}}_\texttt{i}).\mathit{emit}(\texttt{m}_i, \bar{\texttt{u}}_i); \texttt{e}, n, X, \mathbb{L}) \to (\texttt{e}, n, X, \mathbb{L}).$$

Then $X \cup X_G \nvdash \text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)$. Since $\mathcal{R}(X) \subseteq X$ and the proof system is monotone, we conclude that $\mathcal{R}(X) \cup X_G \nvdash \text{LoggedCall}(n-1, \texttt{C}_i, \bar{\texttt{v}}_i, \texttt{m}_i, \bar{\texttt{u}}_i)$.

It then implies that

$$(\texttt{new C}_\texttt{i}(\bar{\texttt{v}}_\texttt{i}).\mathit{emit}(\texttt{m}_i, \bar{\texttt{u}}_i); \texttt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)) \to (\texttt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)).$$

□

Theorem 3.4.1 *Let* $(\texttt{e}, n, X, \mathbb{L}) \to^* (\texttt{e}', n', X', \mathbb{L}')$ *in* $\mathrm{FJ}_{\mathrm{log}}$. *This implies that*

$$(\texttt{e}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X)) \to^* (\texttt{e}', n', \mathcal{R}(X'), \mathbb{L}', \mathcal{W}(X'))$$

in $\mathrm{FJ}'_{\mathrm{log}}$.

Proof. It is straightforward by induction on the derivation of multi-step evaluation $(\texttt{e}, n, X, \mathbb{L}) \to^* (\texttt{e}', n', X', \mathbb{L}')$ using the result of Lemma 3.4.8. □

The following corollary states the correctness in the sense that a program could be evaluated in $\mathrm{FJ}'_{\mathrm{log}}$ with reduced set of logging preconditions and the same audit log as it is evaluated in $\mathrm{FJ}_{\mathrm{log}}$.

Corollary 3.4.2 *If* $(\texttt{e}, 0, \emptyset, \emptyset) \to^* (\texttt{v}, n, X, \mathbb{L})$ *in* $\mathrm{FJ}_{\mathrm{log}}$, *then*

$$(\texttt{e}, 0, \emptyset, \emptyset, \emptyset) \to^* (\texttt{v}, n, \mathcal{R}(X), \mathbb{L}, \mathcal{W}(X))$$

in $\mathrm{FJ}'_{\mathrm{log}}$.

3.4.3 An Illustrative Example for Memory Overhead Mitigation

In the following example, we demonstrate details of our formulation for mitigating memory overhead, considering a sample logging specification.

Example 3.4.1 *In this example, for the sake of brevity we assume that FJ includes primitive values e.g., integers as objects. Consider the logging specification in Figure 3.18. Then,*

$$\Psi = \{\text{Call}(t_i, \mathtt{C}_i, \mathtt{x}_i, \mathtt{m}_i, \mathtt{y}_i) \mid i \in 0 \cdots 4\} \cup \{t_1 < t_2, \mathtt{y}_1 \% 2 = 0, \mathtt{y}_3 = \mathtt{y}_4,$$
$$\text{HasSecLevel}(\mathtt{y}_4, \textbf{secret}), \text{Includes}(\mathtt{y}_0, \mathtt{y}_4)\}$$

and obviously $FV(\Psi) = \{t_i, \mathtt{x}_i, \mathtt{y}_i \mid i \in 0 \cdots 4\}$. Note that each literal could be defined intensionally or extensionally beside the guideline.

Then, the relation $\circledast_{FV}$ includes several pairs including $(t_1, t_2), (\mathtt{y}_3, \mathtt{y}_4), (\mathtt{y}_4, \mathtt{y}_0)$ and $(t_i, \mathtt{y}_i)$ for all $i \in \{0, \cdots, 4\}$ among other pairs.

We then have the following equivalence classes:

$$C_1 = \{t_1, \mathtt{x}_1, \mathtt{y}_1, t_2, \mathtt{x}_2, \mathtt{y}_2\} \qquad C_2 = \{t_0, \mathtt{x}_0, \mathtt{y}_0, t_3, \mathtt{x}_3, \mathtt{y}_3, t_4, \mathtt{x}_4, \mathtt{y}_4\}$$

Note that $\omega(1) = \omega(2) = 1$ and $\omega(0) = \omega(3) = \omega(4) = 2$. As an example, $\bar{\bar{t}}(2) = t_0, t_3, t_4$ is a possible sequence of timestamp variables of class C_2.

This implies the following predicate classes in Ψ:

$P_{C_1} = \{\text{Call}(t_1, \texttt{C}_1, \texttt{x}_1, \texttt{m}_1, \texttt{y}_1), \text{Call}(t_2, \texttt{C}_2, \texttt{x}_2, \texttt{m}_2, \texttt{y}_2), t_1 < t_2, \texttt{y}_1\%2 = 0\}$ $P_{C_2} = \{\texttt{y}_3 = \texttt{y}_4, \text{HasSecLevel}(\texttt{y}_4, \textbf{secret}), \text{Includes}(\texttt{y}_0, \texttt{y}_4)\} \cup_{i=0,3,4} \{\text{Call}(t_i, \texttt{C}_i, \texttt{x}_i, \texttt{m}_i, \texttt{y}_i)\}$

Then, Q_1 and Q_2 could be defined accordingly.

$$Q_1 = \text{Call}(t_1, \texttt{C}_1, \texttt{x}_1, \texttt{m}_1, \texttt{y}_1) \wedge \text{Call}(t_2, \texttt{C}_2, \texttt{x}_2, \texttt{m}_2, \texttt{y}_2) \wedge t_1 < t_2 \wedge \texttt{y}_1\%2 = 0,$$

$$\begin{aligned} Q_2 = &\text{Call}(t_0, \texttt{C}_1, \texttt{x}_0, \texttt{m}_0, \texttt{y}_0) \wedge \text{Call}(t_3, \texttt{C}_3, \texttt{x}_3, \texttt{m}_3, \texttt{y}_3) \wedge \text{Call}(t_4, \texttt{C}_4, \texttt{x}_4, \texttt{m}_4, \texttt{y}_4) \wedge \texttt{y}_3 = \texttt{y}_4 \wedge \\ &\text{HasSecLevel}(\texttt{y}_4, \textbf{secret}) \wedge \text{Includes}(\texttt{y}_0, \texttt{y}_4). \end{aligned}$$

Let $X = \{\text{Call}(2, \texttt{C}_1, 0, \texttt{m}_1, 4)\}$ and $W = \emptyset$. We know that $Q_{[t_2]} = Q_1$ and $Q_{[t_4]} = Q_2$. We then have

$$X \cup \{\text{Call}(6, \texttt{C}_2, 0, \texttt{m}_2, 5)\} \cup X_G \vdash Q_{[t_2]}\big[2/t_1\big]\big[4/\texttt{y}_1\big]\big[6/t_2\big]\big[5/\texttt{y}_2\big]\big[0/\texttt{x}_1\big]\big[0/\texttt{x}_1\big].$$

Therefore, according to Precondition-6 we would have

$$\begin{aligned} &(\texttt{callEvent}(\texttt{m}_2, 5); \texttt{e}, 7, \{\text{Call}(2, \texttt{C}_1, 0, \texttt{m}_1, 4)\}, \mathbb{L}, \emptyset) \to \\ &\qquad (\texttt{e}, 7, \{\text{Call}(2, \texttt{C}_1, 0, \texttt{m}_1, 4), \text{Call}(6, \texttt{C}_2, 0, \texttt{m}_2, 5)\}, \mathbb{L}, \{\texttt{C}_1.\texttt{m}_1, \texttt{C}_2.\texttt{m}_2\}). \end{aligned}$$

Since a ground form of $Q_{[t_2]}$ is derived, we add both $\texttt{C}_1.\texttt{m}_1$ and $\texttt{C}_2.\texttt{m}_2$ to W to ensure that we will not add any invocation information of these triggers to X, any more.

Now suppose X has grown to be

$$X = \{\mathrm{Call}(2, \mathtt{C}_1, 0, \mathtt{m}_1, 4), \mathrm{Call}(6, \mathtt{C}_2, 0, \mathtt{m}_2, 5),$$
$$\mathrm{Call}(11, \mathtt{C}_3, 0, \mathtt{m}_3, 7), \mathrm{Call}(16, \mathtt{C}_0, 0, \mathtt{m}_0, [5, 7, 9])\}.$$

Note that the argument to $\mathtt{C}_0.\mathtt{m}_0$ *is a list. At time 16,* LoggedCall *is not derivable as one of triggers (*$\mathtt{C}_4.\mathtt{m}_4$*) has not been called yet. We then have*

$$X \cup \{\mathrm{Call}(25, \mathtt{C}_4, 0, \mathtt{m}_4, 7)\} \cup X_G \vdash$$
$$Q_{[t_4]}\Big[11/t_3\Big]\Big[7/\mathtt{y}_3\Big]\Big[16/t_0\Big]\Big[[5, 7, 9]/\mathtt{y}_0\Big]\Big[25/t_4\Big]\Big[7/\mathtt{y}_4\Big]\Big[0/\mathtt{x}_0, \mathtt{x}_3, \mathtt{x}_4\Big],$$

assuming that $X_G \vdash \mathrm{HasSecLevel}(7, \mathbf{secret})$*. Then, according to Precondition-7 we will have*

$$(\mathtt{callEvent}(\mathtt{m}_4, 7); \mathtt{e}, 26, X, \mathbb{L}, \{\mathtt{C}_1.\mathtt{m}_1, \mathtt{C}_2.\mathtt{m}_2\}) \rightarrow$$
$$(\mathtt{e}, 26, X \cup \{\mathrm{Call}(25, \mathtt{C}_4, 0, \mathtt{m}_4, 7)\}, \mathbb{L}, \{\{\mathtt{C}_1.\mathtt{m}_1, \mathtt{C}_2.\mathtt{m}_2\}).$$

Despite the fact that a ground form of $Q_{[t_4]}$ *is derived,* W *is not extended with trigger names, e.g.,* $\mathtt{C}_4.\mathtt{m}_4$*. This helps us adding invocations of these triggers to* X *in the future, which might help derive* LoggedCall *predicates that otherwise we were not able to derive. For instance, in this example, if* $\mathrm{Call}(27, \mathtt{C}_3, 0, \mathtt{m}_3, 2)$*,* $\mathrm{Call}(31, \mathtt{C}_4, 0, \mathtt{m}_4, 2)$*, and* $\mathrm{Call}(34, \mathtt{C}_0, 0, \mathtt{m}_0, [1, 4, 2])$ *are added later to the set of logging preconditions, the predicate* $\mathrm{LoggedCall}(34, \mathtt{C}_0, 0, \mathtt{m}_0, [1, 4, 2])$ *is derivable*[3]*.*

[3] Assuming that $X_G \vdash \mathrm{HasSecLevel}(2, \mathbf{secret})$.

Precondition-1

$$\frac{i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \in W}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X, \mathbb{L}, W)}$$

Precondition-2

$$\frac{i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} = \emptyset}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, n, \mathbb{L}, W \cup \{\mathtt{C}_i.\mathtt{m}_i\})}$$

Precondition-3

$$\frac{\begin{array}{c} i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} \neq \emptyset \\ [t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} = \emptyset \qquad X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\} \cup X_G \nvdash Q_{[t_i]}\left[n-1/t_i\right]\left[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\right]\left[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\right] \end{array}}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X, \mathbb{L}, W)}$$

Precondition-4

$$\frac{\begin{array}{c} i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} \neq \emptyset \\ [t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} = \emptyset \qquad X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\} \cup X_G \vdash Q_{[t_i]}\left[n-1/t_i\right]\left[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\right]\left[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\right] \end{array}}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, \mathbb{L}, W \cup \{\mathtt{C}_i.\mathtt{m}_i\})}$$

Precondition-5

$$\frac{\begin{array}{c} i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} \neq \emptyset \qquad [t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} \neq \emptyset \\ [t_i] \cap \{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\} = \emptyset \qquad \nexists \bar{a}_i \,.\, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i))\} \cup X_G \vdash Q_{[t_i]}\left[\bar{a}_i/\bar{\alpha}_i\right]\left[n-1/t_i\right]\left[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\right]\left[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\right] \end{array}}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, \mathbb{L}, W)}$$

Precondition-6

$$\frac{\begin{array}{c} i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} \neq \emptyset \qquad [t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} \neq \emptyset \\ [t_i] \cap \{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\} = \emptyset \qquad \exists \bar{a}_i \,.\, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i))\} \cup X_G \vdash Q_{[t_i]}\left[\bar{a}_i/\bar{\alpha}_i\right]\left[n-1/t_i\right]\left[\bar{\mathtt{v}}_i/\bar{\mathtt{x}}_i\right]\left[\bar{\mathtt{u}}_i/\bar{\mathtt{y}}_i\right] \end{array}}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, \mathbb{L}, W \cup_{t_{i'} \in [t_i]} \{\mathtt{C}_{i'}.\mathtt{m}_{i'}\})}$$

Precondition-7

$$\frac{i \in 1 \cdots n \qquad \mathtt{C}_i.\mathtt{m}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathtt{C}_i, \bar{\mathtt{x}}_i, \mathtt{m}_i, \bar{\mathtt{y}}_i)\} \neq \emptyset \qquad [t_i] - \{\bar{\mathtt{x}}_i, \bar{\mathtt{y}}_i, t_i\} \neq \emptyset \qquad [t_i] \cap \{\bar{\mathtt{x}}_0, \bar{\mathtt{y}}_0, t_0\} \neq \emptyset}{(\mathtt{new\ C_i}(\bar{\mathtt{v}}_\mathtt{i}).\mathtt{callEvent}(\mathtt{m}_i, \bar{\mathtt{u}}_i); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_i, \bar{\mathtt{v}}_i, \mathtt{m}_i, \bar{\mathtt{u}}_i)\}, \mathbb{L}, W)}$$

Precondition-8

$$\frac{}{(\mathtt{new\ C_0}(\bar{\mathtt{v}}_0).\mathtt{callEvent}(\mathtt{m}_0, \bar{\mathtt{u}}_0); \mathtt{e}, n, X, \mathbb{L}, W) \to (\mathtt{e}, n, X \cup \{\mathrm{Call}(n-1, \mathtt{C}_0, \bar{\mathtt{v}}_0, \mathtt{m}_0, \bar{\mathtt{u}}_0)\}, \mathbb{L}, W)}$$

Figure 3.17: Precondition rules for $\mathrm{FJ}'_{\mathrm{log}}$.

$$\forall t_0, \ldots, t_4, \mathtt{x}_0, \ldots, \mathtt{x}_4, \mathtt{y}_0, \ldots, \mathtt{y}_4 \,.\, \mathrm{Call}(t_0, \mathtt{C}_0, \mathtt{x}_0, \mathtt{m}_0, \mathtt{y}_0) \bigwedge_{i=1}^{4} (\mathrm{Call}(t_i, \mathtt{C}_i, \mathtt{x}_i, \mathtt{m}_i, \mathtt{y}_i) \wedge t_i < t_0) \wedge$$
$$t_1 < t_2 \wedge \mathtt{y}_1 \% 2 = 0 \wedge \mathtt{y}_3 = \mathtt{y}_4 \wedge \mathrm{HasSecLevel}(\mathtt{y}_4, \mathbf{secret}) \wedge \mathrm{Includes}(\mathtt{y}_0, \mathtt{y}_4) \implies \mathrm{LoggedCall}(t_0, \mathtt{C}_0, \mathtt{x}_0, \mathtt{m}_0, \mathtt{y}_0).$$

Figure 3.18: Logging Specification for Example 3.4.1

Chapter 4

Direct Information Flow: Dynamic Integrity Taint Analysis

In this chapter, we focus on another realm of application for combining prospective and retrospective measures. As introduced in Chapter 1, retrospective measures could be leveraged as a testing suite to study the potentially vulnerable design and implementation of prospective controls. For example, granted accesses to users can be audited at runtime to discover potential vulnerabilities in the access control policy specification and enforcement. Another example in this application space, is employing in-depth policy specification and enforcement to mitigate imperfect input sanitization in taint analysis regarding direct flow of data integrity. Input sanitization refers to the analysis and potential modification of user provided data in order to enhance data integrity. However, erroneous implementation of input sanitizers leave systems susceptible to command injection vulnerabilities that could be exploited by different attacks, e.g., XSS, SQL injection, etc. As discussed in Section 1.3, many real-world systems suffer from programming bugs that provide erroneous data sanitization. By in-depth enforcement of taint analysis such vulnerabilities are discovered at runtime, which helps mitigating injection attacks in a later stage.

In this chapter, we discuss how we can provably enforce prospective and retrospective measures in order to ameliorate legacy code against injection attacks [85]. In this regard, we use the language model specified in Section 3.1 with some extensions. In particular, we add the support for primitive types that may convey user supplied data. The details of these extensions are discussed in Section 4.1. We use FOL-based specifications to define taint and its propagation in each step of computation. Using these specifications, we define in-depth policies that analyze direct data integrity flow both prospectively and retrospectively at runtime. These definitions are given in Section 4.2. The rewriting algorithm that enforces these in-depth policies are given in Section 4.3. The operational correctness of prospective enforcement and soundness/completeness of retrospective enforcement are studied in Section 4.4. Later in Chapter 5, we will propose a semantic framework to model direct flow of data integrity, called explicit integrity. Explicit integrity provides an underlying model for dynamic integrity taint analysis, and identifies the security property that integrity taint analysis tools support. This framework could be applied in functional settings as well as other programming paradigms. Using explicit integrity, different taint analysis tools can be studied for correctness purposes. In particular, we prove that our rewriting algorithm satisfies explicit integrity.

4.1 An OO Model for Integrity Taint Analysis

In order to study dynamic integrity taint analysis in FJ, we extend the semantics for library methods that allow specification of operations on base values (such as strings and integers). Consideration of these features is important for a thorough modeling of Phosphor-style taint analysis, and important related issues such as string- vs. character-based taint [26] which have not been considered in previous formal work on taint analysis [32]. Since static analysis is not a topic of this dissertation, for brevity we omit the standard FJ type analysis which is described in [27].

The abstract calculus described in Section 3.1 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 [25], 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 [25]. Clearly this strategy introduces a potential for over-taint. Other systems do not propagate taint from strings to their component characters when decomposed [26], 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. These extensions are given in Figure 4.1. 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. If $\texttt{C.m} \in \mathit{LibMeths}$, class `C` may inherit the library method `m` from another class `D`.

The set of security sensitive operations (*SSOs*) and the set of sanitizer methods (*Sanitizers*), introduced in Section 1.3, are similarly closed under inheritance, if the methods are not overriden. We assume that in the body of each sanitizer, `endorse` method is invoked. This method, simply returns `this` object that has been sanitized.

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. Sim-

$$\frac{\texttt{D.m} \in \mathit{LibMeths} \qquad \text{Inherit}(\texttt{m}, \texttt{C}, \texttt{D})}{\texttt{C.m} \in \mathit{LibMeths}}$$

$$\frac{\texttt{D.m} \in \mathit{SSOs} \qquad \text{Inherit}(\texttt{m}, \texttt{C}, \texttt{D})}{\texttt{C.m} \in \mathit{SSOs}} \qquad\qquad \frac{\texttt{D.m} \in \mathit{Sanitizers} \qquad \text{Inherit}(\texttt{m}, \texttt{C}, \texttt{D})}{\texttt{C.m} \in \mathit{Sanitizers}}$$

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

$$\texttt{f}^* \in \mathit{PrimField} \qquad e ::= \nu \mid \texttt{e.f}^* \qquad \texttt{e} ::= \cdots \mid \mathit{Op}(\bar{e}) \qquad \texttt{v} ::= \texttt{new C}(\bar{\texttt{v}}) \mid \nu \qquad \texttt{E} ::= \cdots \mid \mathit{Op}(\bar{\nu}, \texttt{E}, \bar{e})$$

Figure 4.1: Extending FJ for Dynamic Integrity Taint Analysis.

ilarly, primitive values ν may only occur in special object fields and be manipulated there by any *Op*.

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

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

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:

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

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))); }
```

In our security model, tainted input source is a specified argument to a top-level program, i.e., in $\mathfrak{p}(\theta)$, we assume that θ is supplied by a low integrity source.

4.2 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 4.2.1 *Suppose a method* `m` *of an untainted* `C` *object with no fields is invoked on a pair of tainted s_1 and untainted s_2 strings:*

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

. The proper shadow is:

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

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 (4.3.4).

4.2.1 Taint Tracking as a Logical Trace Property

In order to specify taint tracking, we extend the mapping *toFOL*$(\cdot)$ that interprets FJ traces as sets of logical facts.

Definition 4.2.1 *We redefine toFOL(·) as a mapping on traces and configurations, according to Definition 3.1.1, with the following extension:*

$$\mathit{toFOL}((\mathtt{E}[Op(\bar{\nu})], n)) = \{\mathrm{PrimCall}(n, Op, \bar{\nu}), \mathrm{Context}(n, \mathtt{E})\}.$$

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.

$$sv ::= \texttt{shadow C}(t, \bar{sv}) \mid \delta \qquad se ::= sv \mid se.\texttt{f} \mid se.\texttt{m}(\bar{se}) \mid \texttt{shadow C}(t, \bar{se}) \mid \texttt{C.m}(se) \mid Op(\bar{se})$$

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

Figure 4.2: Shadow Values, Expressions, and Evaluation Contexts.

4.2.2 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. The sytax for shadow values, expressions, and evaluation contexts is given in Figure 4.2.

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 4.2.2 *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 in Figure 4.3, where* $srewrite(\bar{\texttt{e}})$ *denotes a mapping of srewrite over the vector* $\bar{\texttt{e}}$.

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

Figure 4.3: Definition of Mapping srewrite.

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 4.4.

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

Figure 4.4: match *Predicate Definition.*

Next, in Figure 4.5, we define a predicate Shadow(n, se) where se is the relevant shadow expression at execution step n, establishing an ordering for the shadow trace. Shadow has as its precondition a "current" shadow expression, and as its postcondition the shadow expression for the next step of evaluation (with the exception of the rule for shadowing *Op*s on primitive values which reflects the "immediate" valuation due to the definition of $\approx$– note the timestamp is not incremented in the postcondition in that case). We set the shadow of the initial configuration at timestamp 1, and then 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.[1]

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

[1]Some notational liberties are taken in Figure 4.5 regarding expression and context substitutions, which are defined using predicates elided for brevity.

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

Figure 4.5: Shadow *Predicate Definition.*

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:

$$\mathrm{meet}(t, t_1, t_2) \Rightarrow \mathrm{Prop}(t, \texttt{String.concat}(t_1, t_2)) \qquad (4.1)$$

4.2.3 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 4.6 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 4.6.

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

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

Figure 4.6: Predicates for Specifying Prospective and Retrospective Properties

Definition 4.2.3 *Let X be the set of rules in Figures 4.4, 4.5, and 4.6 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.*

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

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

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

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

Definition 4.2.4 *We call a program $\mathfrak{p}(\theta)$* safe *iff for all τ it is the case that $\mathfrak{p}(\theta) \Downarrow \tau$ implies $\tau \in \text{SP}_{\text{taint}}$. We call the program* unsafe *iff there exists some trace τ such that $\mathfrak{p}(\theta) \Downarrow \tau$ and $\tau \notin \text{SP}_{\text{taint}}$.*

4.3 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.

4.3.1 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 $\mathtt{e};\mathtt{e}$ to target language expressions, and evaluation contexts of the form $\mathtt{E};\mathtt{e}.$ along with the appropriate operational semantics rule that we define below in Section 4.3.3.

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 blocks if the argument is tainted, and an `endorse` method for any object class returned by a sanitizer.

As discussed in Chapter 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 4.2.3, we define `endorse` so it takes object taint t to the join of t and $\odot$. The algorithm also adds a `log` method call to the beginning of *SSOs*, which will log objects that are maybe tainted or worse. The semantics of `log` are defined directly in the operational semantics of $\mathrm{FJ}_{\mathrm{taint}}$ below.

Definition 4.3.1 *For any expression* $\mathtt{e}$*, the expression* $\mu(\mathtt{e})$ *is syntactically equivalent to* $\mathtt{e}$ *except with every subexpression* $\mathtt{new\ C}(\bar{\mathtt{e}})$ *replaced with* $\mathtt{new\ C}(\circ, \bar{\mathtt{e}})$*. Given SSOs and Sanitizers, define* $\mathcal{R}(\mathtt{e}, CT) = (\mu(\mathtt{e}), \mathcal{R}(CT))$*, where* $\mathcal{R}(CT)$ *is the smallest class table satisfying the axioms given in Figure 4.7.*

4.3.2 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 [25]. We assume that library methods are instrumented to propagate taint as intended (i.e. in accordance with the user defined predicate Prop).

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

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

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

Figure 4.7: Axioms for Rewriting Algorithm

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))); }
```

4.3.3 Operational Semantics of FJ$_{\text{taint}}$

To support the semantics of `log`, we add an audit log $\mathbb{L}$ as a new configuration component in FJ$_{\text{taint}}$ 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. These extensions are given in Figure 4.8.

Reduce
$$\frac{(\mathtt{e}, n) \to (\mathtt{e}', n')}{(\mathtt{e}, n, \mathbb{L}) \to (\mathtt{e}', n'\mathbb{L})}$$

Sequence
$$(\mathtt{v}; \mathtt{e}, n) \to (\mathtt{e}, n)$$

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

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

Figure 4.8: Operational Semantics of FJ$_{\text{taint}}$.

As for FJ we use $\to^*$ to denote the reflexive, transitive closure on $\to$ over FJ$_{\text{taint}}$ configurations of the form $(\mathtt{e}, n, \mathbb{L})$. We define FJ$_{\text{taint}}$ configurations and traces as for FJ. Abusing notation, we write $\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \tau$ iff τ begins with the configuration $(\mathcal{R}(\mathfrak{p}(\theta)), 1, \varnothing)$, and also $\kappa \Downarrow \tau$ iff τ is a valid trace in the FJ$_{\text{taint}}$ semantics beginning with κ.

4.3.4 An Illustrative Example: Dynamic Integrity Taint Analysis

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 4.3.1 *Let*

$$\begin{aligned} &\mathit{mbody}_{CT}(\mathtt{main}, \mathtt{TopLevel}) = \\ &\quad \mathtt{x}, \mathtt{new\ Sec().secureMeth(new\ Sec().sanitize(x.concat(} \\ &\qquad \mathtt{new\ String(''world'')))}. \end{aligned}$$

```
𝔭(new String(″hello ″))
  → TopLevel.main(new Sec().secureMeth(new Sec().sanitize(
            new String(″hello ″).concat(new String(″world″))))
  → TopLevel.main(new Sec().secureMeth(new Sec().sanitize(
            String.concat(new String(@(new String(″hello ″).val, new String(″world″).val))))))
  → TopLevel.main(new Sec().secureMeth(new Sec().sanitize(
            String.concat(new String(@(″hello ″, new String(″world″).val))))))
  → TopLevel.main(new Sec().secureMeth(new Sec().sanitize(String.concat(new String(″hello world″)))))
  → TopLevel.main(new Sec().secureMeth(new Sec().sanitize(new String(″hello world″))))
  → TopLevel.main(new Sec().secureMeth(Sec.sanitize(new String(″hello world″).endorse())))
  → TopLevel.main(new Sec().secureMeth(Sec.sanitize(String.endorse(new String(″hello world″)))))
  → TopLevel.main(new Sec().secureMeth(Sec.sanitize(new String(″hello world″))))
  → TopLevel.main(new Sec().secureMeth(new String(″hello world″)))
  → TopLevel.main(Sec.secureMeth(new String(″hello world″)))
  → TopLevel.main(new String(″hello world″))
  → new String(″hello world″).
```

Figure 4.9: Example 4.3.1: Source Trace.

Assume the string ″`hello` ″ *is tainted with low integrity. Figure 4.9, Figure 4.10, and Figure 4.11 give the source trace, the shadow expressions and the target trace, respectively. Note that shadow expressions in Figure 4.10 are derived based on the rules given in Figure 4.5. For the sake of brevity and clarity in illustrating the main ideas, we have assumed that methods* `Sec.sanitize` *and* `Sec.secureMeth` *are identity functions. Some reduction steps are elided in the example as n-length multi-step reductions* $\rightarrow^n$.

4.4 Properties of Program Rewriting

The logical definition of in-depth integrity taint analysis presented in Section 4.2 establishes the proper specification of prospective and retrospective analysis. In this

$$\begin{array}{l}
\text{Shadow}(1, \texttt{shadow TopLevel}(\circ)\texttt{.main}(\texttt{shadow String}(\bullet, \delta)))\\
\text{Shadow}(2, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\\
\quad \texttt{shadow String}(\bullet, \delta)\texttt{.concat}(\texttt{shadow String}(\circ, \delta))))))\\
\text{Shadow}(3, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\\
\quad \texttt{String.concat}(\texttt{shadow String}(\bullet, @(\texttt{shadow String}(\bullet, \delta)\texttt{.val}, \texttt{shadow String}(\circ, \delta)\texttt{.val})))))))\\
\text{Shadow}(4, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\\
\quad \texttt{String.concat}(\texttt{shadow String}(\bullet, @(\delta, \texttt{shadow String}(\circ, \delta)\texttt{.val})))))))\\
\text{Shadow}(5, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\\
\quad \texttt{String.concat}(\texttt{shadow String}(\bullet, @(\delta, \delta)))))))\\
\text{Shadow}(5, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\\
\quad \texttt{String.concat}(\texttt{shadow String}(\bullet, \delta))))))\\
\text{Shadow}(6, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow Sec}(\circ)\texttt{.sanitize}(\texttt{shadow String}(\bullet, \delta))))))\\
\text{Shadow}(7, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{Sec.sanitize}(\texttt{shadow String}(\bullet, \delta)\texttt{.endorse}()))))\\
\text{Shadow}(8, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{Sec.sanitize}(\texttt{String.endorse}(\texttt{shadow String}(\bullet, \delta))))))\\
\text{Shadow}(9, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{Sec.sanitize}(\texttt{shadow String}(\bullet, \delta)))))\\
\text{Shadow}(10, \texttt{TopLevel.main}(\texttt{shadow Sec}(\circ)\texttt{.secureMeth}(\texttt{shadow String}(\odot, \delta))))\\
\text{Shadow}(11, \texttt{TopLevel.main}(\texttt{Sec.secureMeth}(\texttt{shadow String}(\odot, \delta))))\\
\text{Shadow}(12, \texttt{TopLevel.main}(\texttt{shadow String}(\odot, \delta)))\\
\text{Shadow}(13, \texttt{shadow String}(\odot, \delta))
\end{array}$$

Figure 4.10: Example 4.3.1: Shadow Expressions.

section, we show how these definitions are used to establish operational correctness of prospective enforcement and soundness/completeness of retrospective enforcement for $\mathcal{R}$, and how these conditions are proven. The main properties defined in this section establish operational correctness for prospective measure and soundness/completeness of retrospective measure in Definitions 4.4.3 and 4.4.5 respectively, and the main results demonstrate that these properties are enjoyed by $\mathcal{R}$ in Theorems 4.4.2 and 4.4.3.

4.4.1 Semantics Preservation

A core condition for operational correctness of $\mathcal{R}$ is the proof of semantics preservation for safe programs in FJ, i.e. that rewritten programs simulate the semantics of

$$
\begin{array}{l}
\mathcal{R}(\mathfrak{p}(\texttt{new String}(''\texttt{hello }''))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new Sec}(\circ)\texttt{.sanitize(} \\
\quad \texttt{new String}(\bullet, ''\texttt{hello }'')\texttt{.concat(new String}(\circ, ''\texttt{world}''))))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new Sec}(\circ)\texttt{.sanitize(} \\
\quad \texttt{String.concat(new String}(\bullet, @(\texttt{new String}(\bullet, ''\texttt{hello }'')\texttt{.val}, \texttt{new String}(\circ, ''\texttt{world}'')\texttt{.val})))))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new Sec}(\circ)\texttt{.sanitize(} \\
\quad \texttt{String.concat(new String}(\bullet, @(''\texttt{hello }'', \texttt{new String}(\circ, ''\texttt{world}'')\texttt{.val})))))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new Sec}(\circ)\texttt{.sanitize(} \\
\quad \texttt{String.concat(new String}(\bullet, ''\texttt{hello world}''))))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new Sec}(\circ)\texttt{.sanitize(new String}(\bullet, ''\texttt{hello world}'')))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(Sec.sanitize(new String}(\bullet, ''\texttt{hello world}'')\texttt{.endorse()}))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(Sec.sanitize(String.endorse(new String}(\odot, ''\texttt{hello world}''))))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(Sec.sanitize(new String}(\odot, ''\texttt{hello world}'')))), \varnothing \\
\to \texttt{TopLevel.main(new Sec}(\circ)\texttt{.secureMeth(new String}(\odot, ''\texttt{hello world}''))), \varnothing \\
\to \texttt{TopLevel.main(Sec.secureMeth(new Sec}(\circ)\texttt{.log(new String}(\odot, ''\texttt{hello world}'')); \\
\quad \texttt{new Sec}(\circ)\texttt{.check(new String}(\odot, ''\texttt{hello world}'')); \texttt{new String}(\odot, ''\texttt{hello world}''))), \varnothing \\
\to \texttt{TopLevel.main(Sec.secureMeth(new Sec}(\circ)\texttt{.check(new String}(\odot, ''\texttt{hello world}'')); \\
\quad \texttt{new String}(\odot, ''\texttt{hello world}''))), \{\texttt{new String}(\odot, ''\texttt{hello world}'')\} \\
\to \texttt{TopLevel.main(Sec.secureMeth(new String}(\odot, ''\texttt{hello world}''))), \{\texttt{new String}(\odot, ''\texttt{hello world}'')\} \\
\to \texttt{TopLevel.main(new String}(\odot, ''\texttt{hello world}'')), \{\texttt{new String}(\odot, ''\texttt{hello world}'')\} \\
\to \texttt{new String}(\odot, ''\texttt{hello world}''), \{\texttt{new String}(\odot, ''\texttt{hello world}'')\}
\end{array}
$$

Figure 4.11: Example 4.3.1: Target Trace.

source program modulo security instrumentations. The way this simulation is defined will naturally imply a full and faithful implementation of taint shadowing semantics. Adapting Definition 2.5.1, 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 4.4.1 *Rewriting algorithm $\mathcal{R}$ is* semantics preserving *iff for all safe programs $\mathfrak{p}(\theta)$ (Definition 4.2.4) 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 4.4.1 $\mathcal{R}(\mathfrak{p}(\theta)) = \mathcal{R}(\mathfrak{p})(\mathcal{R}(\theta))$

4.4.2 Operational Correctness for Prospective Analysis

Proof of semantics preservation establishes operational 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 4.4.2 *An FJ_{taint} program* $\mathtt{e}$ causes a security failure *iff*

$$\mathtt{e}, 1, \varnothing \to^* \mathtt{E}[\mathtt{v.check}(\mathtt{new\ C}(\bullet, \bar{\mathtt{v}}))], n, \mathbb{L}$$

for some $\mathtt{E}$*,* $\mathtt{v}$*,* $\mathtt{new\ C}(\bullet, \bar{\mathtt{v}})$*, n, and* $\mathbb{L}$*.*

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

Definition 4.4.3 *We call rewriting algorithm $\mathcal{R}$ operationally correct for prospective enforcement provided that a program $\mathfrak{p}(\theta)$ is unsafe (Definition 4.2.4) iff $\mathcal{R}(\mathfrak{p}(\theta))$ causes a security failure (Definition 4.4.2).*

4.4.3 Soundness/Completeness 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 4.4.4 *For FJ_{taint} programs we write $\tau \rightsquigarrow \mathbb{L}$ iff $tail(\sigma) = \texttt{e}, n, \mathbb{L}$ 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 soundness/completeness conditions regarding retrospective enforcement for rewriting algorithms. Note that satisfaction of either of these conditions only implies condition (1) of Definition 4.4.1, not condition (2), so semantics preservation is an independent condition. We define $toFOL(\mathbb{L}) = \{\text{MaybeBAD}(\texttt{v}) \mid \texttt{v} \in \mathbb{L}\}$, and thus $\lfloor \mathbb{L} \rfloor = C(toFOL(\mathbb{L}))$.

Definition 4.4.5 *Rewriting algorithm $\mathcal{R}$ is* retrospectively sound/complete *for retrospective enforcement iff for all programs $\mathcal{R}(\mathfrak{p}(\theta))$, and finite traces τ and σ where:*

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

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

4.4.4 Definition of :≈ and Correctness Results

To establish operational correctness of prospective enforcement and soundness/completeness of retrospective enforcement for the program rewriting algorithm, we need to define a correspondence relation $:\approx$. Source language execution traces and target language execution traces correspond if they represent the same expression evaluated to the same point. We make a special case: when a sink method is called in the source execution, in which the target execution needs to first check the arguments to the sink method in order to `log` and enforce prospective policy by `check`. In this case, 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 4.4.6 *The mapping overlay* $: (\mathtt{e}, se) \mapsto \mathtt{e}'$ *is defined in Figure 4.12.*

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.

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

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

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

Figure 4.12: Definition of overlay.

Considering the rule

$$\text{Shadow}(n, se) \implies \text{LShadow}(se), \tag{4.2}$$

we define $\mathit{LastShadow}(\tau) = se$ such that $\lfloor\tau\rfloor \otimes X \vdash \text{LShadow}(se)$, where X contains the rules given in Figure 4.4, Figure 4.5 and (4.2). 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 4.4.2 *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 4.5. □

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` and `log`.

Definition 4.4.7 *The mapping trim* : $\mathtt{e} \mapsto \mathtt{e}'$ *is defined in Figure 4.13. We assume* ϵ *to be no-op, i.e.,* $\epsilon; \mathtt{e} = \mathtt{e}$.

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

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

$$\mathit{trim}(\mathtt{e.m}(\bar{\mathtt{e}}')) = \begin{cases} \epsilon & \text{if } \mathtt{m} \in \{\mathtt{log}, \mathtt{check}\} \\ \mathit{trim}(\mathtt{e}).\mathtt{m}(\overline{\mathit{trim}(\mathtt{e}')}) & \text{if } \mathtt{m} \notin \{\mathtt{log}, \mathtt{check}\} \end{cases}$$

Figure 4.13: Definition of trim.

Definition 4.4.8 *Given source language execution trace $\tau = \sigma\kappa$ and target language execution trace $\tau' = \sigma'\kappa'$, $\tau :\approx \tau'$ iff $\mathit{overlay}(\kappa, \mathit{LastShadow}(\tau)) = \mathit{trim}(\mathtt{e}')$, where $\kappa' = \mathtt{e}', n', \mathbb{L}$.*

In what follows, we prove the semantics preservation given by Definition 4.4.1. To this end, in Lemma 4.4.3, 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 4.4.4 states that if *trim* of a non-security failure expression $\mathtt{e}$ is reduced to $\mathtt{e}'$ then $\mathtt{e}$ reduces (in potentially multiple steps) to some expression with the same *trim* as $\mathtt{e}'$.

Lemma 4.4.3 *For all expressions $\mathtt{e}$, if $\mathit{trim}(\mathtt{e}) = \mathtt{v}$, then either (1) for all $\mathbb{L}$ there exists some trace σ such that $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ where $\kappa = \mathtt{v}, n', \mathbb{L}'$ for some n' and $\mathbb{L}'$, or (2) $\mathtt{e}$ causes a security failure.*

Proof. By induction on the structure of $\mathtt{e}$. Most of the cases are trivial. Non-trivial cases are as follows:

- Let $\mathtt{e} = \mathtt{new\ C}(\mathtt{e}_1, \cdots, \mathtt{e}_n)$ where $\mathtt{e}$ does not cause a security failure. We need to show that if $\mathit{trim}(\mathtt{new\ C}(\mathtt{e}_1, \cdots, \mathtt{e}_n)) = \mathit{trim}(\mathtt{v})$, then for all n and $\mathbb{L}$ there exist some trace σ such that $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ where $\kappa = \mathtt{v}, n', \mathbb{L}'$ for some n' and $\mathbb{L}'$. Let $\mathtt{v} = \mathtt{new\ C}(\mathtt{v}_1, \cdots, \mathtt{v}_n)$. Since $\mathit{trim}(\mathtt{new\ C}(\mathtt{e}_1, \cdots, \mathtt{e}_n)) = \mathtt{new\ C}(\mathit{trim}(\mathtt{e}_1), \cdots, \mathit{trim}(\mathtt{e}_n))$, for each $i \in \{1, \cdots n\}$ we have $\mathit{trim}(\mathtt{e}_i) = \mathtt{v}_i$. Then by induction hypothesis, for all n_i and $\mathbb{L}_i$ there exists some trace σ_i such that $\mathtt{e}_i, n_i, \mathbb{L}_i \Downarrow \sigma_i\kappa_i$ where $\kappa_i = \mathtt{v}_i, n_i', \mathbb{L}_i'$ for some n_i' and $\mathbb{L}_i'$. This straightforwardly gives the required result.

- Let $\mathtt{e} = \mathtt{e}_1; \mathtt{e}_2$ where $\mathtt{e}$ does not cause a security failure. We need to show that if $\mathit{trim}(\mathtt{e}_1; \mathtt{e}_2) = \mathit{trim}(\mathtt{v})$, then for all n and $\mathbb{L}$ there exist some trace σ such that $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ where $\kappa = \mathtt{v}, n', \mathbb{L}'$ for some n' and $\mathbb{L}'$. Since $\mathit{trim}(\mathtt{e}_1, \mathtt{e}_2) = \mathit{trim}(\mathtt{e}_1); \mathit{trim}(\mathtt{e}_2)$, there are two cases: (1) $\mathit{trim}(\mathtt{e}_1) = \epsilon$ and $\mathit{trim}(\mathtt{e}_2) = \mathtt{v}$, or (2) $\mathit{trim}(\mathtt{e}_2) = \epsilon$ and $\mathit{trim}(\mathtt{e}_1) = \mathtt{v}$. In either case the result is immediate by induction hypothesis.

□

Lemma 4.4.4 *For all expressions* $\mathtt{e}$*, if* $\mathit{trim}(\mathtt{e}), n, \mathbb{L} \rightarrow \mathtt{e}', n', \mathbb{L}'$ *then either (1) there exists* σ *such that* $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ *with* $\kappa = \mathtt{e}'', n'', \mathbb{L}''$ *and* $\mathit{trim}(\mathtt{e}'') = \mathit{trim}(\mathtt{e}')$*, or (2)* $\mathtt{e}$ *causes a security failure.*

Proof. By induction on the structure of $\mathtt{e}$, and applying Lemma 4.4.3. □

Lemma 4.4.5 states that *overlay*ing a method body with its shadow is equal to the same method body in the rewritten class table.

Lemma 4.4.5 $\mathit{overlay}(\mathtt{e}, \mathit{srewrite}(\mathtt{e})) = \mu(\mathtt{e})$*.*

Proof. By induction on the structure of $\mathtt{e}$. Most of the cases are immediate or concluded from the induction hypothesis. The core case is where $\mathtt{e} = \mathtt{new\ C}(\bar{\mathtt{e}})$. In this case, we have $\mathit{overlay}(\texttt{new C}(\bar{\texttt{e}}), \mathit{srewrite}(\texttt{new C}(\bar{\texttt{e}}))) = \mathit{overlay}(\texttt{new C}(\bar{\texttt{e}}), \texttt{shadow C}(\circ, \bar{\texttt{e}}))$. Using the induction hypothesis, $\overline{\mathit{overlay}(\texttt{e}, \mathit{srewrite}(\texttt{e}))} = \overline{\mu(\texttt{e})}$, we then have

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

□

Lemma 4.4.6 and Lemma 4.4.7 state that single step and multi step reductions in FJ preserve $:\approx$.

Lemma 4.4.6 *If* $\tau_1\texttt{e}_1 :\approx \tau_2\kappa_2$ *and* $\texttt{e}_1 \rightarrow \texttt{e}_1'$ *then there exists* σ *such that* $\kappa_2 \Downarrow \sigma$ *and* $\tau_1\texttt{e}_1\texttt{e}_1' :\approx \tau_2\sigma$.

Proof. It is proven by induction on the derivation of $\texttt{e}_1 \rightarrow \texttt{e}_1'$ and applying Lemmas 4.4.4 and 4.4.5. In the following the interesting cases are studied.

- Let $\texttt{new C}(\bar{\texttt{v}}).\texttt{f}_i \rightarrow \texttt{v}_i$, i.e., $\texttt{e}_1 = \texttt{new C}(\bar{\texttt{v}})$ and $\texttt{e}_1' = \texttt{v}_i$. Then the shadow of $\texttt{e}_1$ is $\mathit{se}_1 = \texttt{shadow C}(t, \bar{sv}).\texttt{f}_i$. Let $\kappa_2 = \texttt{e}_2, n_2, \mathbb{L}_2$. Then, according to $\tau_1\texttt{e}_1 :\approx \tau_2\kappa_2$, we have $\mathit{trim}(\texttt{e}_2) = \mathit{overlay}(\texttt{e}_1, \mathit{se}_1) = \texttt{new C}(t, \overline{\mathit{overlay}(\texttt{v}, sv)}).\texttt{f}_i$. Then, $\mathit{trim}(\texttt{e}_2), n_2, \mathbb{L}_2 \rightarrow \mathit{overlay}(\texttt{v}_i, sv_i), n_2, \mathbb{L}_2$. Using Lemma 4.4.4, there exists σ such that $\kappa_2 \Downarrow \sigma\kappa$ with $\kappa = \texttt{e}'_2, n_2', \mathbb{L}_2'$ and $\mathit{trim}(\texttt{e}'_2) = \mathit{overlay}(\texttt{v}_i, sv_i)$, which implies that $\tau_1\texttt{e}_1\texttt{e}_1' :\approx \tau_2\sigma\kappa$.

- Let $\texttt{C.m}(\texttt{v}) \rightarrow \texttt{v}$, i.e., $\texttt{e}_1 = \texttt{C.m}(\texttt{v})$ and $\texttt{e}_1' = \texttt{v}$. Then the shadow of $\texttt{e}_1$ is $\mathit{se}_1 = \texttt{C.m}(sv)$. Let $\kappa_2 = \texttt{e}_2, n_2, \mathbb{L}_2$. Then, according to $\tau_1\texttt{e}_1 :\approx \tau_2\kappa_2$, we have $\mathit{trim}(\texttt{e}_2) = \mathit{overlay}(\texttt{e}_1, \mathit{se}_1) = \texttt{C.m}(\mathit{overlay}(\texttt{v}, sv))$. Then, $\mathit{trim}(\texttt{e}_2), n_2, \mathbb{L}_2 \rightarrow \mathit{overlay}(\texttt{v}, sv), n_2, \mathbb{L}_2$. Using Lemma 4.4.4, there exists σ such that $\kappa_2 \Downarrow \sigma\kappa$ with $\kappa = \texttt{e}'_2, n_2', \mathbb{L}_2'$ and $\mathit{trim}(\texttt{e}'_2) = \mathit{overlay}(\texttt{v}, sv)$, which implies that $\tau_1\texttt{e}_1\texttt{e}_1' :\approx \tau_2\sigma\kappa$.

- Let $\texttt{v.m}(\bar{\texttt{u}}) \to \texttt{e}[\texttt{v}/\texttt{this}][\bar{\texttt{u}}/\bar{\texttt{x}}]$, where $\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e}$. That is, $\texttt{e}_1 = \texttt{v.m}(\bar{\texttt{u}})$ and $\texttt{e}_1' = \texttt{e}[\texttt{v}/\texttt{this}][\bar{\texttt{u}}/\bar{\texttt{x}}]$. Then the shadow of $\texttt{e}_1$ is $se_1 = sv.\texttt{m}(\bar{su})$. Let $\kappa_2 = \texttt{e}_2, n_2, \mathbb{L}_2$. Then, according to $\tau_1\texttt{e}_1 :\approx \tau_2\kappa_2$, we have $\mathit{trim}(\texttt{e}_2) = \mathit{overlay}(\texttt{e}_1, se_1) = \mathit{overlay}(\texttt{v}, sv).\texttt{m}(\overline{\mathit{overlay}(\texttt{u}, su)})$. We consider the following cases for $\texttt{C.m}$:

  - If $\texttt{C.m} \in \mathit{SSOs}$ then $\mathit{trim}(\texttt{e}_2), n_2, \mathbb{L}_2 \to \hat{\texttt{e}}, n_2, \mathbb{L}_2$, where

    $$\hat{\texttt{e}} = \texttt{this.log(x)}; \texttt{this.check(x)}; \mu(\texttt{e})[\mathit{overlay}(\texttt{v}, sv)/\mathit{this}][\mathit{overlay}(\texttt{u}, su)/x].$$

    Using Lemma 4.4.4, there exists σ such that $\kappa_2 \Downarrow \sigma\kappa$ with $\kappa = \texttt{e}'_2, n_2', \mathbb{L}_2'$ and $\mathit{trim}(\texttt{e}'_2) = \mathit{trim}(\hat{\texttt{e}})$. We have

    $$\begin{aligned}\mathit{trim}(\hat{\texttt{e}}) &= \mathit{trim}(\texttt{this.log(x)}; \texttt{this.check(x)}; \\ &\qquad \mu(\texttt{e})[\mathit{overlay}(\texttt{v}, sv)/\mathit{this}][\mathit{overlay}(\texttt{u}, su)/x]) \\ &= \mathit{trim}(\mu(\texttt{e}[\mathit{overlay}(\texttt{v}, sv)/\mathit{this}][\mathit{overlay}(\texttt{u}, su)/x])) \\ &= \mu(\texttt{e})[\mathit{overlay}(\texttt{v}, sv)/\mathit{this}][\mathit{overlay}(\texttt{u}, su)/x].\end{aligned}$$

    According to Lemma 4.4.5, $\mathit{overlay}(\texttt{e}, se) = \mu(\texttt{e})$. Thus,

    $$\begin{aligned}\mathit{trim}(\texttt{e}'_2) = \mathit{trim}(\hat{\texttt{e}}) &= \mathit{overlay}(\texttt{e}, se)[\mathit{overlay}(\texttt{v}, sv)/\mathit{this}][\mathit{overlay}(\texttt{u}, su)/x] \\ &= \mathit{overlay}(\texttt{e}[\texttt{v}/\texttt{this}][\texttt{u}/\texttt{x}], se[sv/\texttt{this}][su/\texttt{x}]),\end{aligned}$$

    which implies that $\tau_1\texttt{e}_1\texttt{e}_1' :\approx \tau_2\sigma\kappa$.

  - If $\texttt{C.m} \notin \mathit{SSOs} \cup \mathit{LibMeths}$, then $\mathit{trim}(\texttt{e}_2), n_2, \mathbb{L}_2 \to \hat{\texttt{e}}, n_2, \mathbb{L}_2$, where $\hat{\texttt{e}} =$

$\mu(\mathtt{e})[\mathit{overlay}(\mathtt{v}, \mathit{sv})/\mathit{this}][\overline{\mathit{overlay}(\mathtt{u}, \mathit{su})}/\bar{\mathtt{x}}]$. Then simialr to the case where $\mathtt{C.m} \in \mathit{SSOs}$, we conclude that $\tau_1 \mathtt{e}_1 \mathtt{e}'_1 :\approx \tau_2 \sigma \kappa$.

– If $\mathtt{C.m} \in \mathit{LibMeths}$ then $\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}, \mathtt{new\ D}(\bar{\mathtt{e}})$. We have $\mathtt{e'}_1 = \mathtt{new\ D}(\bar{\mathtt{e}})[\mathtt{v}/\mathtt{this}][\bar{\mathtt{u}}/\bar{\mathtt{x}}]$ and $\mathit{trim}(\mathtt{e}_2), n_2, \mathbb{L}_2 \to \hat{\mathtt{e}}, n_2, \mathbb{L}_2$, where

$$\hat{\mathtt{e}} = \mathtt{new\ D}(t, \bar{\mathtt{e}})[\mathit{overlay}(\mathtt{v}, \mathit{sv})/\mathit{this}][\overline{\mathit{overlay}(\mathtt{u}, \mathit{su})}/\bar{\mathtt{x}}]$$

such that $\mathrm{Prop}(t, \mathtt{C.m}(t_0 \wedge \bar{t}))$, t_0 is the taint tag in sv and $\bar{t}$ is the sequence of taint tag in $\overline{\mathit{su}}$. Using Lemma 4.4.4, there exists σ such that $\kappa_2 \Downarrow \sigma\kappa$ with $\kappa = \mathtt{e'}_2, n'_2, \mathbb{L}'_2$ and $\mathit{trim}(\mathtt{e'}_2) = \mathit{trim}(\hat{\mathtt{e}})$. We have

$$\begin{aligned}
\mathit{trim}(\mathtt{e'}_2) &= \mathit{trim}(\hat{\mathtt{e}}) \\
&= \mathtt{new\ D}(t, \overline{\mathit{trim}(\mathtt{e})})[\mathit{overlay}(\mathtt{v}, \mathit{sv})/\mathit{this}][\overline{\mathit{overlay}(\mathtt{u}, \mathit{su})}/\bar{\mathtt{x}}] \\
&= \mathtt{new\ D}(t, \bar{\mathtt{e}})[\mathit{overlay}(\mathtt{v}, \mathit{sv})/\mathit{this}][\overline{\mathit{overlay}(\mathtt{u}, \mathit{su})}/\bar{\mathtt{x}}] \\
&= \mathit{overlay}(\mathtt{new\ D}(\bar{\mathtt{e}})[\bar{\mathtt{u}}/\bar{\mathtt{x}}][\mathtt{v}/\mathtt{this}], \mathtt{shadow\ D}(t, \overline{\mathit{se}})[\overline{\mathit{su}}/\bar{\mathtt{x}}][\mathit{sv}/\mathtt{this}]),
\end{aligned}$$

which implies that $\tau_1 \mathtt{e}_1 \mathtt{e}'_1 :\approx \tau_2 \sigma \kappa$.

□

Lemma 4.4.7 *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 $\mathtt{e}_1 \Downarrow \sigma_1$ (reflexive and transitive closure) and applying Lemma 4.4.6. □

Similarly, Lemma 4.4.8 and Lemma 4.4.9 argue that single step and multi step reductions in $\mathrm{FJ}_{\mathrm{taint}}$ preserve $:\approx$.

Lemma 4.4.8 *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. It is proven by induction on the derivation of $\kappa_2 \rightarrow \kappa_2'$ and applying Lemma 4.4.5. The interesting case is method invocation. Let $\kappa_2 = \mathtt{e}_2, n_2, \mathbb{L}_2$, where $\mathtt{e}_2 = \mathtt{v.m}(\bar{\mathtt{u}})$. Assume that $\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}, \mathtt{e}$. Then, we have the following two cases:

- If $\mathtt{C.m} \in \mathit{SSOs}$ then $\mathtt{e}_2 \rightarrow \mathtt{log(u)}; \mathtt{check(u)}; \mu(\mathtt{e})[\mathtt{u}/\mathtt{x}][\mathtt{v}/\mathtt{this}]$. Since $\tau_1 \mathtt{e}_1 :\approx \tau_2 \kappa_2$, we have $\mathit{trim}(\mathtt{e}_2) = \mathit{overlay}(\mathtt{e}_1, \mathit{se}_1)$, which implies that $\mathtt{e}_1 = \mathtt{v}'.\mathtt{m}(\mathtt{u}')$ such that $\mathit{overlay}(\mathtt{v}', \mathit{sv}') = \mathtt{v}$ and $\mathit{overlay}(\mathtt{u}', \mathit{su}') = \mathtt{u}$. By Lemma 4.4.5, we conclude that

  $$\mathit{overlay}(\mathtt{e}[\mathtt{u}'/\mathtt{x}][\mathtt{v}'/\mathtt{this}], \mathit{se}[\mathit{su}'/\mathtt{x}][\mathit{sv}'/\mathtt{this}]) = \mathit{trim}(\mathtt{log(u)}; \mathtt{check(u)}; \mu(\mathtt{e})[\mathtt{u}/\mathtt{x}][\mathtt{v}/\mathtt{this}]).$$

  As $\mathtt{e}_1 \rightarrow \mathtt{e}[\mathtt{u}'/\mathtt{x}][\mathtt{v}'/\mathtt{this}]$, the proof is complete for this case.

- If $\mathtt{C.m} \notin \mathit{SSOs}$ then $\mathtt{e}_2 \rightarrow \mu(\mathtt{e})[\bar{\mathtt{u}}/\bar{\mathtt{x}}][\mathtt{v}/\mathtt{this}]$. Since $\tau_1 \mathtt{e}_1 :\approx \tau_2 \kappa_2$, we have $\mathit{trim}(\mathtt{e}_2) = \mathit{overlay}(\mathtt{e}_1, \mathit{se}_1)$, which implies that $\mathtt{e}_1 = \mathtt{v}'.\mathtt{m}(\bar{\mathtt{u}}')$ such that $\mathit{overlay}(\mathtt{v}', \mathit{sv}') = \mathtt{v}$ and $\overline{\mathit{overlay}(\mathtt{u}', \mathit{su}')} = \bar{\mathtt{u}}$. By Lemma 4.4.5, we conclude that

  $$\mathit{overlay}(\mathtt{e}[\bar{\mathtt{u}'}/\bar{\mathtt{x}}][\mathtt{v}'/\mathtt{this}], \mathit{se}[\overline{\mathit{su}'}/\bar{\mathtt{x}}][\mathit{sv}'/\mathtt{this}]) = \mathit{trim}(\mu(\mathtt{e})[\bar{\mathtt{u}}/\mathtt{x}][\mathtt{v}/\mathtt{this}]).$$

  As $\mathtt{e}_1 \rightarrow \mathtt{e}[\bar{\mathtt{u}'}/\mathtt{x}][\mathtt{v}'/\mathtt{this}]$, the proof is complete for this case similar the case above.

$\square$

Lemma 4.4.9 *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 $\kappa_2 \Downarrow \sigma_2$ (reflexive and transitive closure) and applying Lemma 4.4.8. □

Lemma 4.4.10 states that initial configuration in $\mathrm{FJ}_{\mathrm{taint}}$ corresponds to the initial configuration in FJ. Finally, in Theorem 4.4.1, the semantics preservation property is proven.

Lemma 4.4.10 *Let $\theta = \mathtt{new}\ \mathtt{C}(\bar{\nu})$ be a tainted input. Then,*

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

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

Theorem 4.4.1 establishes semantics preservation for rewriting algorithm $\mathcal{R}$.

Theorem 4.4.1 *The rewriting algorithm $\mathcal{R}$ is semantics preserving (Definition 4.4.1).*

Proof. Lemma 4.4.10 states that initial configuration of a program $\mathfrak{p}$ corresponds to the initial configuration of $\mathcal{R}(\mathfrak{p})$. Lemmas 4.4.7 and 4.4.9 extend the correspondence relation for traces of arbitrary lengths. More specifically, Lemmas 4.4.10 and 4.4.7 entail the 1st condition of Definition 4.4.1, and Lemmas 4.4.10 and 4.4.9 result in the 2nd condition of Definition 4.4.1. □

In order to prove that prospective component of $\mathcal{R}$ is operationally correct, we first need to show that $\mathrm{SP}_{\mathrm{taint}}$ is a safety property.

Lemma 4.4.11 $\mathrm{SP}_{\mathrm{taint}}$ *is a safety property.*

Proof. Let $\tau \notin \mathrm{SP}_{\mathrm{taint}}$. Then, $(\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\mathrm{BAD}\}} \neq C(\varnothing)$. This implies that there exists some n such that $\mathrm{BAD}(n) \in (\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow\{\mathrm{BAD}\}}$. Let $\tau[\cdots n]$ denote the finite prefix of τ up to timestamp n. By Definition 4.2.3 BAD only refers to events that precede step n, so it follows that $\lfloor\tau\rfloor \otimes C(X) \vdash \mathrm{BAD}(n)$ iff $\lfloor\tau[\cdots n]\rfloor \otimes C(X) \vdash \mathrm{BAD}(n)$, i.e. $\tau \notin \mathrm{SP}_{\mathrm{taint}}$ iff $\tau[\cdots n] \notin \mathrm{SP}_{\mathrm{taint}}$ for finite n, hence $\mathrm{SP}_{\mathrm{taint}}$ is a safety property [1]. $\square$

Theorem 4.4.2 *The rewriting algorithm $\mathcal{R}$ is operationally correct for prospective enforcement (Definition 4.4.3).*

Proof. Suppose on the one hand that $\mathfrak{p}(\theta)$ is unsafe, which is the case iff $\mathfrak{p}(\theta) \Downarrow \tau$ and $\tau \notin \mathrm{SP}_{\mathrm{taint}}$ for some τ. Then according to Lemma 4.4.11, there exists some timestamp n such that $\tau[\cdots n]$ characterizes $\mathrm{SP}_{\mathrm{taint}}$, i.e., $\mathrm{BAD}(n)$ is derivable from the rules in Definition 4.2.3 and so $(\tau[\cdots n])\sigma \notin \mathrm{SP}_{\mathrm{taint}}$ for any σ. Let n be the least timestamp with such property and $\tau' = \tau[\cdots n-1]$, and $\tau[n] = \mathtt{E}[\mathtt{v.m}(\mathtt{new\ D}(\bar{\mathtt{u}}))]$ where $\mathtt{v.m}$ is an sso invocation and $\mathtt{new\ D}(\bar{\mathtt{u}})$ has low integrity by the Definition 4.2.3. By Theorem 4.4.1 there exists some trace σ, such that $\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \sigma$ and $\tau' :\approx \sigma$. Therefore, by definition of $:\approx$ and $\rightarrow$ we may assert that $\mathit{tail}(\sigma) \Downarrow \kappa_1\kappa_2\kappa_3$ such that

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

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

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

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

Supposing on the other hand that $\mathcal{R}(\mathfrak{p}(\theta))$ causes a security failure, it follows that $\mathfrak{p}(\theta)$ is unsafe by similar reasoning (i.e. fundamental appeal to Theorem 4.4.1), since security `check`s are only added to the beginning of *SSOs* and fail if the argument has low integrity. □

In what follows, we give a proof for retrospective soundness/completeness of rewriting algorithm $\mathcal{R}$, given in Definition 4.4.5. As in Section 3.2, 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 [86]). In essence, we demonstrate that audit logs generated by $\mathrm{FJ}_{\mathrm{taint}}$ programs *are* the least Herbrand model of the logging specification for the source program, hence contain the same information.

We study the cases where a record is added to the audit log for single step (Lemma 4.4.12) and multi step (Lemma 4.4.13) reductions.

Lemma 4.4.12 *Let* $\mathtt{e}, n, \mathbb{L} \to \mathtt{e}', n', \mathbb{L}'$. *If* $\mathtt{v} \in \mathbb{L}' - \mathbb{L}$ *then we have* $\mathtt{e} = \mathtt{E}[\mathtt{u.log(v)}]$ *for some evaluation context* $\mathtt{E}$ *and value* $\mathtt{u}$*,* $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ *and* $t \leq \odot$*.*

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

Lemma 4.4.13 *Let* $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$*, where* $\kappa = \mathtt{e}', n', \mathbb{L}'$*. 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* $\kappa_1 = \mathtt{e}_1, n_1, \mathbb{L}_1$*,* $\kappa_2 = \mathtt{e}_2, n_2, \mathbb{L}_2$*, where* $\mathtt{e}_1 = \mathtt{E}[\mathtt{u.log(v)}]$ *for some evaluation context* $\mathtt{E}$ *and value* $\mathtt{u}$*,* $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ *and* $t \leq \odot$*.*

Proof. By induction on the derivation of $\mathtt{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ and applying Lemma 4.4.12. □

Lemmas 4.4.14, 4.4.15 and 4.4.16 extend the results of Lemmas 4.4.4, 4.4.6 and 4.4.7 respectively, for $\mathrm{FJ}_{\mathrm{taint}}$ traces with maximal length.

Lemma 4.4.14 *For all expressions* e*, if* $\mathit{trim}(\mathsf{e}), n, \mathbb{L} \rightarrow \mathsf{e}', n', \mathbb{L}'$ *then either (1) there exists* σ *such that* $\mathsf{e}, n, \mathbb{L} \Downarrow \sigma\kappa$ *with* $\kappa = \mathsf{e}'', n'', \mathbb{L}''$*,* $\mathit{trim}(\mathsf{e}'') = \mathit{trim}(\mathsf{e}')$ *and* $\kappa \rightarrow \hat{\mathsf{e}}, \hat{n}, \hat{\mathbb{L}}$ *for some* $\hat{\mathsf{e}}$ $\hat{n}$*, and* $\hat{\mathbb{L}}$ *implies* $\mathit{trim}(\hat{\mathsf{e}}) \neq \mathit{trim}(\mathsf{e}')$ *or (2)* e *causes a security failure.*

Proof. By induction on the structure of e, and applying Lemma 4.4.3. □

Lemma 4.4.15 *If* $\tau_1 \mathsf{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathsf{e}_1 \rightarrow \mathsf{e}_1'$ *then there exists* σ *such that* $\kappa_2 \Downarrow \sigma$*,* $\tau_1\mathsf{e}_1\mathsf{e}_1' :\approx \tau_2\sigma$*, and if* $\mathit{tail}(\sigma) = \kappa_2'$ *and* $\kappa_2' \rightarrow \kappa_2''$ *for some* κ_2'' *then* $\tau_1\mathsf{e}_1\mathsf{e}_1' :\not\approx \tau_2\sigma\kappa_2''$.

Proof. By induction on the derivation of $\mathsf{e}_1 \rightarrow \mathsf{e}_1'$ and applying Lemmas 4.4.14 and 4.4.5. □

Lemma 4.4.16 *If* $\tau_1\mathsf{e}_1 :\approx \tau_2\kappa_2$ *and* $\mathsf{e}_1 \Downarrow \sigma_1$ *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* $\mathit{tail}(\sigma_2) = \kappa_2'$ *and* $\kappa_2' \rightarrow \kappa_2''$ *for some* κ_2'' *then* $\tau_1\sigma_1 :\not\approx \tau_2\sigma_2\kappa_2''$.

Proof. By induction on the derivation of $\mathsf{e}_1 \Downarrow \sigma_1$ and applying Lemma 4.4.15. □

In Lemma 4.4.17, 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 soundness/completeness in Theorem 4.4.3.

Lemma 4.4.17 *Given* $\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \sigma$ *and* $\tau :\approx \sigma$ *and* $\sigma \rightsquigarrow \mathbb{L}$*, we have:*

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

Proof. (Sketch) We first show that $\mathit{toFOL}(\mathbb{L})$ is a subset of $\mathfrak{H}(X \cup \mathit{toFOL}(\tau)) \cap L_{\{\mathrm{MaybeBAD}\}}$. Let $\mathrm{MaybeBAD}(\mathtt{v}) \in \mathit{toFOL}(\mathbb{L})$. According to the definition of $\mathit{toFOL}(\mathbb{L})$, $\mathtt{v} \in \mathbb{L}$. Using Lemma 4.4.13, there exists some trace $\sigma'\kappa_1\kappa_2$ as the prefix of σ such that $\kappa_1 = \mathtt{e}_1, n_1, \mathbb{L}_1$, $\kappa_2 = \mathtt{e}_2, n_2, \mathbb{L}_2$, where $\mathtt{e}_1 = \mathtt{E}[\mathtt{u.log}(\mathtt{v})]$ for some evaluation context $\mathtt{E}$ and value $\mathtt{u}$, $\mathtt{v} = \mathtt{new\ C}(t, \overline{\mathtt{v}'})$ and $t \leq \odot$. Using Theorem 4.4.1, there exists a trace $\hat{\tau}$ such that $\mathfrak{p}(\theta) \Downarrow \hat{\tau}$ and $\hat{\tau} :\approx \sigma'\kappa_1\kappa_2$. This ensures that

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

as $\mathtt{u.log}(\mathtt{v})$ could only appear in the body of some method $\mathtt{C.m} \in \mathit{SSOs}$, according to the rewriting algorithm $\mathcal{R}$, and thus the preconditions of the last rule defined in Figure 4.6 are satisfied by $X \cup \mathit{toFOL}(\hat{\tau})$. Moreover, $\hat{\tau}$ is a prefix of τ, and thus $\mathit{toFOL}(\hat{\tau}) \subseteq \mathit{toFOL}(\tau)$. This entails that

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

Next, we show that $\mathfrak{H}(X \cup \mathit{toFOL}(\tau)) \cap L_{\{\mathrm{MaybeBAD}\}}$ is a subset of $\mathit{toFOL}(\mathbb{L})$. Let $\mathrm{MaybeBAD}(\mathtt{new\ D}(t', \bar{\mathtt{v}}')) \in \mathfrak{H}(X \cup \mathit{toFOL}(\tau)) \cap L_{\{\mathrm{MaybeBAD}\}}$ and $\mathtt{v} = \mathtt{new\ D}(\bar{\mathtt{v}}')$. Then, there exist some n, $\mathtt{C}$, $\bar{\mathtt{u}}$ and $\mathtt{m}$ where $\mathrm{Call}(n, \mathtt{C}, \bar{\mathtt{u}}, \mathtt{m}, \mathtt{v}) \in \mathit{toFOL}(\tau)$. Moreover, there exist SE, se, t, $\overline{se}$ and $\overline{se'}$, where the predicates $\mathrm{Shadow}(n, se)$ and $\mathrm{match}(se, SE, \mathtt{shadow\ C}(t, \overline{se}).\mathtt{m}(\mathtt{shadow\ D}(t', \overline{se'})))$ are derivable from the rules in X, $\mathtt{C.m} \in \mathit{SSOs}$ and $t' \leq \odot$. Let $\tau[\cdots n]$ denote the prefix of τ ending in timestamp n. Based on the definition of $\mathit{toFOL}(\cdot)$, we can infer that $\mathit{tail}(\tau[\cdots n]) = \mathtt{E}[\mathtt{new\ C}(\bar{\mathtt{u}}).\mathtt{m}(\mathtt{v})]$. Using Theorem 4.4.1, we know that there exists trace σ' such that $\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \sigma'$ and

$\tau[\cdots n] :\approx \sigma'$. Let $\mathit{tail}(\sigma') = \hat{\mathtt{e}}, \hat{n}, \hat{\mathbb{L}}$. Therefore,

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

Obviously, $\mathit{trim}(\hat{\mathtt{e}}), \hat{n}, \hat{\mathbb{L}} \to \hat{\mathtt{E}}[\mathtt{new}\ \mathtt{C}(t, \bar{\mathtt{u}}).\mathtt{log}(\mathtt{new}\ \mathtt{D}(t', \bar{\mathtt{v}'})); \mathtt{e}], \hat{n}, \hat{\mathbb{L}}$ according to the semantics of target language. Then, using Lemma 4.4.4, $\mathit{tail}(\sigma') \Downarrow \sigma''\kappa_1\kappa_2\kappa_3$, where

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

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

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

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

Theorem 4.4.3 *The rewriting algorithm $\mathcal{R}$ is retrospectively sound and complete (Definition 4.4.5).*

Proof. Let $\mathfrak{p}$ be a source program and LS be a logging specification defined as $LS = \mathit{spec}(X, \{\text{MaybeBAD}\})$. We aim to show that for all $\mathcal{R}(\mathfrak{p}(\theta))$ and finite traces τ and

σ, such that $\mathcal{R}(\mathfrak{p}(\theta)) \Downarrow \sigma$, $\tau :\approx \sigma$ and $\sigma \rightsquigarrow \mathbb{L}$, $C(\mathit{toFOL}(\mathbb{L})) = LS(\tau)$. By Lemma 4.4.17, we have

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

By Lemma 3.2.1 and Lemma 3.2.2

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

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

Chapter 5

The Meaning of Dynamic Integrity Taint Analysis

The semantics of information flow has been well studied and is typically characterized via noninterference properties, but surprisingly little work has been done to develop similar properties for taint analysis. Recently, it has been shown that direct flow of data confidentiality is not comparable with noninterference [24], i.e., there are both noninterfering programs with direct leakage of secret data to public domain, and programs without such direct leakges, but interfering. For instance consider the following two statements in a core imperative language, in which s and p are respectively secret and public variables:

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

The first statement is noninterfering, but direct flow of information from s to p exists, whereas the second statement is interfering due to the indirect flow from s to p, but there are not any direct flows from s to p. In this chapter, we show similar examples that distinguish noninterference from the security property that is enforced by dynamic integrity taint analysis. Our examples are in a functional setting with hierarchical data structures.

Formal definitions of taint analysis implementations do exist, but they are usually operational in nature. For example, in Section 4.4.4, we have established an operational correctness for the prospective enforcement of direct integrity flow.

In this chapter, we propose a semantic framework to model direct flow of data integrity enforced by integrity taint analysis techniques. We intend to use the proposed semantics to establish provable correctness conditions for rewriting algorithms that instrument integrity taint analysis in the presence of input sanitization. This way an underlying semantic framework is provided to study numerous other integrity taint analyzers in the future.

5.1 Direct Integrity Flow Semantics: Explicit Integrity

In this section, we build "explicit integrity" a semantic property that builds on the notions of *explicit secrecy* [24] and *attacker power* [37]. Similar to explicit secrecy, explicit integrity is language-agnostic. In later sections, we discuss instantiation of this model for FJ.

Explicit secrecy is defined as a property of a program, where the execution does not change the knowledge of a low confidentiality user. *Knowledge* [64] is defined as the set of initial states that a low confidentiality user is able to consider to generate a given sequence of observables. *Explicit knowledge* [24] restricts attacker knowledge for direct confidentiality flows only. In this section, we demonstrate how explicit knowledge can be “dualized” for direct integrity flow analysis and applied as a semantic framework for dynamic integrity taint analysis tools, particularly in functional languages with hierarchical data structures (FJ).

Attacker power [37] is introduced as a counterpart to attacker knowledge in the context of integrity, as the set of low integrity inputs that generate the same sequence of high integrity events. Each high integrity event could be a simple assignment to a predefined high integrity variable, a method that manipulates trusted data (secure sinks), etc. according to the language model. The more refined the attacker power is, the more powerful the low integrity attacker becomes, as she becomes more capable to distinguish between the effects of different attacks on high integrity data.

We define *explicit attacker power* as the attacker power constrained on direct integrity flows. Then, *explicit integrity* is defined as the property of preserving explicit attacker power during program execution. In order to limit flows to direct ones, we have followed the techniques introduced in [24] to define *state transformers*. State transformers extract direct flows semantically by specifying the ways in which program state is modified in each step of execution, along with direct-flow events that are generated.

5.1.1 Model Specification

In what follows, we formulate our explicit integrity semantics. We first define the interface for our framework. Let $\mathbf{K}$ be the of program configurations for a given object language. κ ranges over configurations. Configurations consist of control and state segments. Control refers to the code and state refers to data. Let $\mathbf{C}$ be the set of controls with $\mathbf{c}$ ranging over the elements of $\mathbf{C}$. Moreover, let $\mathbf{S}$ denote the set of states and $\mathbf{s}$ represent a given state. We also define a set of high integrity events, $\mathbf{E}$. A high integrity event $\mathbf{e}$ may refer to different computations in different language models and settings. For example, it could be as simple as assigning a low integrity data to a high integrity variable, or invoking a method with a low integrity data as its parameter to store that parameter in a database. We assume the existence of the small step evaluation relation $\rightarrow \subseteq \mathbf{K} \times \mathbf{E}^* \times \mathbf{K}$ where $(\kappa, \overline{\mathbf{e}}, \kappa') \in \rightarrow$ is denoted as $\kappa \xrightarrow{\overline{\mathbf{e}}} \kappa'$. We use $\kappa \rightarrow \kappa'$ if $\overline{\mathbf{e}}$ is empty (ϵ) or could be elided in the discussion. Notation $\rightarrow^*$ is used for reflexive and transitive closure of $\rightarrow$.

Each configuration is considered to include two segments: control (code) and state (data). These segments are not necessarily disjoint and could overlap in some language models. In this regard, let mappings $\mathit{state} : \mathbf{K} \rightarrow \mathbf{S}$ and $\mathit{control} : \mathbf{K} \rightarrow \mathbf{C}$ extract the state and control segments of configurations, and $\langle \cdot, \cdot \rangle : \mathbf{C} \times \mathbf{S} \rightarrow \mathbf{K}$ construct a configuration from its control and state segments. These mappings need to satisfy the following property, for any κ:

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

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

We define state transformers as follows:

Definition 5.1.1 *Let $\kappa \to \kappa'$ and control(κ) $= \mathbf{c}$ for some $\mathbf{c}$. $f : \mathbf{S} \to \mathbf{S} \times \mathbf{E}^*$ is the function where $f(\mathbf{s}) = (\mathit{state}(\kappa''), \overline{\mathbf{e}})$ for all $\mathbf{s}$ and for the unique κ'' and $\overline{\mathbf{e}}$ such that $\langle \mathbf{c}, \mathbf{s} \rangle \xrightarrow{\overline{\mathbf{e}}} \kappa''$. We write $\kappa \to_f \kappa'$ to associate the state transformer f with the reduction $\kappa \to \kappa'$.*

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

We now define the power an attacker obtains by observing high integrity events. We capture this by defining a set of high integrity equivalent states that generate the same sequence of high integrity events. We posit the binary relation $=_{\mathbb{T}}$ on $\mathbf{S}$ to denote high integrity equivalent (or trust equivalent) states. Intuitively, $\mathbf{s} =_{\mathbb{T}} \mathbf{s}'$ if $\mathbf{s}$ and $\mathbf{s}'$ agree on high integrity data. The instantiation of the relation depends on the language model in which the states are defined. For a state $\mathbf{s}$ and some state transformer f, the state $\mathbf{s}'$ is considered as an element of the explicit attacker power, if $\mathbf{s} =_{\mathbb{T}} \mathbf{s}'$ and $\mathbf{s}'$ agrees with $\mathbf{s}$ on the generated high integrity events.

Definition 5.1.2 *We define* explicit attacker power *with respect to state* $\mathbf{s}$ *and state transformer f as follows, where projection on the ith element of a tuple is denoted by π_i.*

$$p_e(\mathbf{s}, f) = \{\mathbf{s}' \mid \mathbf{s} =_{\mathbb{T}} \mathbf{s}', \pi_2(f(\mathbf{s})) = \pi_2(f(\mathbf{s}'))\}.$$

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

Definition 5.1.3 *A control* $\mathbf{c}$ *satisfies* explicit integrity *for state* $\mathbf{s}$*, iff* $\langle \mathbf{c}, \mathbf{s}\rangle \rightarrow_f^* \kappa'$ *implies that for any* $\mathbf{s}'$ *and* $\mathbf{s}''$*, if* $\mathbf{s}' =_{\mathbb{T}} \mathbf{s}''$ *then we have* $\mathbf{s}'' \in p_e(\mathbf{s}', f)$*.*

A control $\mathbf{c}$ *satisfies* explicit integrity *iff for any* $\mathbf{s}$*,* $\mathbf{c}$ *satisfies explicit integrity for* $\mathbf{s}$.

We can now consider explicit integrity in the presence of endorsement in the style of gradual release [64]. We assume that there exists a set of integrity events $\mathbf{E_{en}} \subseteq \mathbf{E}$ that are generated when endorsements occur. Explicit attacker power is only allowed to change for such events.

Definition 5.1.4 *A control* $\mathbf{c}$ *satisfies* explicit integrity modulo endorsement *for state* $\mathbf{s}$ *iff* $\langle \mathbf{c}, \mathbf{s}\rangle \rightarrow_f^* \kappa' \xrightarrow{\overline{\mathbf{e}}}{}_g^* \kappa''$ *and* $\overline{\mathbf{e}} \notin \mathbf{E_{en}}^*$ *imply that* $p_e(\mathbf{s}, f) = p_e(\mathbf{s}, g * f)$*.*

We also define a variant of noninterference for the sake of comparison with explicit integrity.

Definition 5.1.5 *Program* $\mathbf{c}$ *is noninterfering iff for any two trust equivalent states* $\mathbf{s}$ *and* $\mathbf{s}'$*, if* $\langle \mathbf{c}, \mathbf{s}\rangle$ *and* $\langle \mathbf{c}, \mathbf{s}'\rangle$ *generate sequence of events* $\overline{\mathbf{e}}$ *and* $\overline{\mathbf{e}}'$ *respectively, up to a given number of steps of evaluation, then we have* $\overline{\mathbf{e}} = \overline{\mathbf{e}}'$*.*

5.2 An OO Model

For practical purposes we are interested in applications of taint analysis for HLLs, especially Java [85]. Therefore in our formulation we consider a core model of Java, which is based on Featherweight Java (FJ), discussed in Section 4.1. We extend the configurations with states, defined below.

5.2.1 Operational Semantics

Rather than a substitution-based semantics (Section 3.1), we use a stack-based semantics to keep track of program variables in effect and their integrity level. Stack-based semantics provides a framework to distinguish between controls and states in functional settings, where as in substitution-based semantics these components are mingled within the structure of expressions. The separation of data from code is leveraged in our framework to study explicit integrity.

We assume the existence of at least two integrity levels **H** and **L** which denote the high integrity and the low integrity levels, respectively. A stack, denoted by Σ, is a possibly empty sequence of substitutions γ, where each γ is a partial mapping from variables to values.

$$\Sigma ::= \varnothing \mid \gamma :: \Sigma$$

Then we can define a stack-based semantics in a standard fashion as follows. Configurations are expression, stack pairs $(\mathtt{e}, \Sigma)$. The reduction relation is ternary, of the form $\kappa \xrightarrow{\alpha} \kappa'$ where α is a sequence of events– either integrity events *iev* or endorsement events *eev*. The former occurs when an sso is invoked, the latter occurs when a sanitizer returns, and both events are parameterized with the values flowing into the sso or the value just sanitized, respectively. Figure 5.1 defines the stack-based operational semantics for FJ.

Context
$$\frac{(\texttt{e}, \Sigma) \overset{\alpha}{\rightarrow} (\texttt{e}', \Sigma')}{(\texttt{E}[\texttt{e}], \Sigma) \overset{\alpha}{\rightarrow} (\texttt{E}[\texttt{e}'], \Sigma')}$$

Var
$$(\texttt{x}, \gamma :: \Sigma) \rightarrow (\gamma(\texttt{x}), \gamma :: \Sigma)$$

Field
$$\frac{\mathit{fields}_{CT}(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}} \qquad \texttt{f}_i \in \bar{\texttt{f}}}{(\texttt{new C}(\bar{\texttt{v}}).\texttt{f}_i, \Sigma) \rightarrow (\texttt{v}_i, \Sigma)}$$

Invoke
$$\frac{\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e} \qquad \texttt{C.m} \notin \mathit{SSOs}}{(\texttt{new C}(\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}}), \Sigma) \rightarrow (\texttt{C.m}(\texttt{e}), [\texttt{new C}(\bar{\texttt{v}})/\texttt{this}][\bar{\texttt{u}}/\bar{\texttt{x}}] :: \Sigma)}$$

SSO
$$\frac{\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}, \texttt{e} \qquad \texttt{C.m} \in \mathit{SSOs}}{(\texttt{new C}(\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}}), \Sigma) \overset{\overline{iev(\texttt{u})}}{\rightarrow} (\texttt{C.m}(\texttt{e}), [\texttt{new C}(\bar{\texttt{v}})/\texttt{this}][\bar{\texttt{u}}/\bar{\texttt{x}}] :: \Sigma)}$$

Return
$$\frac{\texttt{C.m} \notin \mathit{Sanitizers}}{(\texttt{C.m}(\texttt{v}), \gamma :: \Sigma) \rightarrow (\texttt{v}, \Sigma)}$$

Sanitized
$$\frac{\texttt{C.m} \in \mathit{Sanitizers}}{(\texttt{C.m}(\texttt{v}), \gamma :: \Sigma) \overset{eev(\texttt{v})}{\rightarrow} (\texttt{v}, \Sigma)}$$

Figure 5.1: Stack-based Operational Semantics for FJ.

$$\frac{\texttt{e redex}}{\mathit{control}(\texttt{E}[\texttt{e}], \Sigma) = \texttt{e}} \qquad \frac{\texttt{e redex}}{\mathit{state}(\texttt{E}[\texttt{e}], \Sigma) = (\texttt{E}, \Sigma)} \qquad \langle \texttt{e}, (\texttt{E}, \Sigma) \rangle = (\texttt{E}[\texttt{e}], \Sigma)$$

Figure 5.2: Definitions of control, state and $\langle \cdot, \cdot \rangle$ in FJ.

5.2.2 Model instantiation for FJ

In this section, we instantiate explicit integrity for FJ. First, we define the required interface specified in Section 5.1. Let $\mathbf{C}$ be the set of *redexes*. We define $\mathbf{s}$ to be a pair of an evaluation context and a stack $(\texttt{E}, \Sigma)$. We define κ as the pair of expressions and stacks, i.e., $(\texttt{e}, \Sigma)$. Mappings *control*, *state* and $\langle \cdot, \cdot \rangle$ are defined in Figure 5.2. These definitions satisfy $\langle \mathit{control}(\kappa), \mathit{state}(\kappa) \rangle = \kappa$.

Lemma 5.2.1 *For any FJ configuration κ, we have $\langle \mathit{control}(\kappa), \mathit{state}(\kappa) \rangle = \kappa$.*

Proof. Straightforward based on the definition of mappings in Figure 5.2. $\square$

$$\frac{\texttt{E}_1 = \texttt{E}_2 \qquad \Sigma_1 =_{\mathbb{T}} \Sigma_2}{(\texttt{E}_1, \Sigma_1) =_{\mathbb{T}} (\texttt{E}_2, \Sigma_2)} \qquad\qquad \varnothing =_{\mathbb{T}} \varnothing$$

$$\frac{\mathit{dom}(\gamma_1) = \mathit{dom}(\gamma_2) \qquad \forall \texttt{x} \in \mathit{dom}(\gamma_1).\Gamma(\texttt{x}) = \mathbf{H} \Rightarrow \gamma_1(\texttt{x}) = \gamma_2(\texttt{x}) \qquad \Sigma_1 =_{\mathbb{T}} \Sigma_2}{(\gamma_1 :: \Sigma_1) =_{\mathbb{T}} (\gamma_2 :: \Sigma_2)}$$

Figure 5.3: Definition trust equivalent states in FJ.

The set $\mathbf{E}$ is the set of events of the form $\mathit{iev}(\texttt{v})$ and $\mathit{eev}(\texttt{v})$. We modify the operational semantics of FJ slightly in order to consider integrity events. When an sso is invoked with argument $\texttt{v}$ the integrity event $\mathit{iev}(\texttt{v})$ is generated. Moreover, the event $\mathit{eev}(\texttt{v})$ is generated when a sanitizer is returned (with endorsed value).

Now, in order to define trust equivalent states in FJ, we posit a function Γ that maps every program variable (method argument) to high or low integrity– without loss of generality we assume that all method variables are uniquely named. Essentially, $\Gamma(\texttt{x}) = \mathbf{H}$ for all and only sso arguments. To complete the instantiation of the model we define trust equivalence relation $=_{\mathbb{T}}$ on FJ states as given in Figure 5.3. Note that we abuse the notation to define trust equivalent stacks as part of the definition. Two states in FJ are trust equivalent, if the evaluation contexts of those states are syntactically equal and their stacks are trust equivalent. Two stacks are trust equivalent if they map high integrity variables to the same values in each activation record.

The initial state is $\mathbf{s}_0 = ([\,], [\mathbf{a}/\texttt{attack}] :: \varnothing)$, where $\Gamma(\texttt{attack}) = \mathbf{L}$. $\mathbf{s}_0$ refers to the initial attacker capabilities.

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

For the sake of brevity, we may elide stacks from FJ-configurations in the following.

5.2.3 Sanity Conditions on Library Methods

As noted before, taint is propagated by library methods according to a user-defined predicate Prop. We define two sanity conditions for library methods: *not undertainting* and *not overtainting*. The former condition is required in the implementation in order to meet explicit integrity modulo endorsement, whereas the latter is a good practice in the implementation of taint analysis tools.

First we define trust equivalence relation between FJ_{taint} configurations in order to specify sanity conditions for library methods. We accomplish this by definition of a relation $=_\circ$ on configurations that requires identity of high integrity values, but allows low integrity values to be different.

Definition 5.2.1 *The relation $=_\circ$ on configurations is the least relation inductively defined by inference rules in Figure 5.4.*

Definition 5.2.2 *Let the following hold:*

- $\texttt{E}[\texttt{new C}(t, \bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}})], \Sigma_1 \rightarrow^* \texttt{E}[\texttt{w}], \Sigma_2,$
- $\texttt{E}'[\texttt{new C}(t', \bar{\texttt{v}}').\texttt{m}(\bar{\texttt{u}}')], \Sigma_1' \rightarrow^* \texttt{E}'[\texttt{w}'], \Sigma_2',$
- $\texttt{C.m} \in$ *LibMeths, and*
- $\texttt{E}[\texttt{new C}(t, \bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}})]], \Sigma_1 =_\circ \texttt{E}'[\texttt{new C}(t', \bar{\texttt{v}}').\texttt{m}(\bar{\texttt{u}}')], \Sigma_1'.$

We say C.m is not undertainting *iff* $\texttt{E}[\texttt{w}], \Sigma_2 =_\circ \texttt{E}'[\texttt{w}'], \Sigma_2'$.

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

$$\frac{\bar{\mathtt{e}}_1 =_\circ \bar{\mathtt{e}}_2}{\mathtt{new\ C}(\circ, \bar{\mathtt{e}}_1) =_\circ \mathtt{new\ C}(\circ, \bar{\mathtt{e}}_2)} \qquad \mathtt{new\ C}(\bullet, \bar{\mathtt{e}}_1) =_\circ \mathtt{new\ D}(\bullet, \bar{\mathtt{e}}_2) \qquad \frac{\mathtt{e}_1 =_\circ \mathtt{e}_1' \qquad \mathtt{e}_2 =_\circ \mathtt{e}_2'}{\mathtt{e}_1; \mathtt{e}_2 =_\circ \mathtt{e}_1'; \mathtt{e}_2'}$$

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

$$\frac{\begin{array}{c}\Sigma_1 =_\circ \Sigma_2 \qquad dom(\gamma_1) = dom(\gamma_2) \qquad \forall \mathtt{x} \in dom(\gamma_1).\gamma_1(\mathtt{x}) = \mathtt{new\ C}(\circ, \bar{\mathtt{v}}) \Rightarrow \bar{\mathtt{v}} =_\circ \bar{\mathtt{u}} \wedge \gamma_2(\mathtt{x}) = \mathtt{new\ C}(\circ, \bar{\mathtt{u}}) \\ \forall \mathtt{x} \in dom(\gamma_1).\gamma_1(\mathtt{x}) = \mathtt{new\ C}(\bullet, \bar{\mathtt{v}}) \Rightarrow \gamma_2(\mathtt{x}) = \mathtt{new\ D}(\bullet, \bar{\mathtt{u}})\end{array}}{\gamma_1 :: \Sigma_1 =_\circ \gamma_2 :: \Sigma_2}$$

$$\frac{\mathtt{e}_1 =_\circ \mathtt{e}_2 \qquad \Sigma_1 =_\circ \Sigma_2}{\mathtt{e}_1, \Sigma_1 =_\circ \mathtt{e}_2, \Sigma_2}$$

Figure 5.4: Definition of $=_\circ$ Relation.

Definition 5.2.3 *Let* `C.m` *be a library methods that is constant. We say* `C.m` is not overtainting *iff* $\text{Prop}(\circ, \mathtt{C.m}(t_1, \overline{t_2}))$.

For example, `String.concat` is not undertainting if the taint propagation policy is defined as the rule (4.1) in Section 4.2.2, but if it is defined as the predicate $\text{Prop}(\circ, \mathtt{String.concat}(t_1, t_2))$ then this library method is undertainting. As an example of overtainting, consider primitive operation $prim(str)$ to be a constant function over strings with return value 0. Let the OO wrapper for *prim* be a library method `String.primwrapper` such that $mbody_{CT}(\mathtt{primwrapper}, \mathtt{String}) = \varnothing, \mathtt{new\ Int}(prim(\mathtt{this.val}))$ with propagation policy $\text{Prop}(t, \mathtt{String.primwrapper}(t))$.

5.2.4 Illustrative Examples: Incompatibility of Noninterference and Explicit Integrity

In what follows, we give examples to show incomparability of noninterference and explicit integrity in FJ. In this regard, Example 5.2.1 demonstrates a program which satisfies explicit integrity while it is interfering. Conversely, the program given in Example 5.2.2 satisfies noninterference but not explicit integrity. Example 5.2.3 discusses a case where the program satisfies explicit integrity modulo endorsement. In the following examples, method $\texttt{C.sso} \in \mathit{SSOs}$ is assumed to be an identity function. Moreover, the shadow of $\mathbf{a}$ is denoted sa.

Example 5.2.1 *Let* $\mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$*,* $\texttt{C} <: \texttt{D}$*,* $\mathit{mbody}_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{new B().sso(new D(}''\texttt{hi}''\texttt{))}$ *and* $\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{new B().sso(new C(}''\texttt{hi}''\texttt{))}$*. Whether the user input to* `TopLevel.main` *is an object of class* `C` *or* `D`*, different methods* `m` *are invoked by dynamic dispatch. Let the attack be* $\texttt{new C(}''\texttt{hello}''\texttt{)}$*. Corresponding execution trace, state transformer definitions, and attacker powers are given in Figure 5.5. Then, according to the composition of state transformers, the attacker power is preserved. Thus, explicit integrity is satisfied. However, this program does not satisfy noninterference. Considering trust equivalent initial states depicted in Figure 5.5, the attacks* $\texttt{new C(}''\texttt{hello}''\texttt{)}$ *and* $\texttt{new D(}''\texttt{hello}''\texttt{)}$ *generate unequal sequence of events* $[\mathit{iev}(\texttt{new C(}''\texttt{hi}''\texttt{)})]$ *and* $[\mathit{iev}(\texttt{new D(}''\texttt{hi}''\texttt{)})]$*.*

Example 5.2.2 *Let* $\mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$*,* $\texttt{C} <: \texttt{D}$*,* $\mathit{mbody}_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{new B().sso(new C())}$ *and* $\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{new B().sso(x)}$*. Note that we assume no fields for classes* `C` *and* `D`*, and so only objects* `new C()` *and* `new D()` *are definable. Let*

$$
\begin{aligned}
&\texttt{new TopLevel().main(new C(}''\texttt{hello}''\texttt{))} \rightarrow^*_{f_1} \\
&\quad \texttt{TopLevel.main(C.m(new B().sso(new C(}''\texttt{hi}''\texttt{))))} \rightarrow^*_{f_2} \texttt{new C(}''\texttt{hi}''\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
&f_1(\texttt{E}, \gamma :: \Sigma) = ((\texttt{E[TopLevel.main(C.m[\,])]}, \\
&\quad [\gamma(\texttt{attack})/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: [\texttt{new TopLevel()}/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: \gamma :: \Sigma), \epsilon) \\
&f_2(\texttt{E[TopLevel.main(C.m[\,])]}, \gamma' :: \gamma :: \Sigma) = ((\texttt{E}, \Sigma), \mathit{iev}(\texttt{new C(}''\texttt{hi}''\texttt{)})) \\
&f_2 * f_1(\mathbf{s}_0) = (\mathbf{s}_0, \mathit{iev}(\texttt{new C(}''\texttt{hi}''\texttt{)}))
\end{aligned}
$$

$$
p_e(\mathbf{s}_0, f_1) = p_e(\mathbf{s}_0, f_2 * f_1) = \{\mathbf{s}' \mid \mathbf{s}' =_{\mathbb{T}} \mathbf{s}_0\}
$$

$$
([\,], [\texttt{new C(}''\texttt{hello}''\texttt{)}/\texttt{attack}] :: \varnothing) =_{\mathbb{T}} ([\,], [\texttt{new D(}''\texttt{hello}''\texttt{)}/\texttt{attack}] :: \varnothing)
$$

Figure 5.5: Example 5.2.1: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States.

the attack be `new C()`. *Similar to Example 5.2.1, details are given in Figure 5.6. Since the explicit attacker power is refined to the attack, explicit integrity is not satisfied. However, this program satisfies noninterference. The attacks* `new C()` *and* `new D()` *both generate the same event* $\mathit{iev}(\texttt{new C()})$ *considering trust equivalent initial states given in Figure 5.6.*

Example 5.2.3 *Let* $\mathit{mbody}_{CT}(\texttt{main}, \texttt{TopLevel}) = \texttt{x}, \texttt{x.m(x)}$, $\texttt{C} <: \texttt{D}$, $\mathit{mbody}_{CT}(\texttt{m}, \texttt{D}) = \texttt{x}, \texttt{new B().sso(new D(}''\texttt{hi}''\texttt{))}$, $\mathit{mbody}_{CT}(\texttt{m}, \texttt{C}) = \texttt{x}, \texttt{new B().sso(new F().sanitize(x))}$, *and* $\mathit{mbody}_{CT}(\texttt{sanitize}, \texttt{F}) = \texttt{x}, \texttt{x.endorse()}$. *Let the attack be* `new C(″hello″)`. *Details are given in Figure 5.7. As explicit attacker power is not further refined after endorsement, explicit integrity modulo endorsement is satisfied. However, the program is not satisfying noninterference, since the attacks* `new C(″hello″)` *and* `new D(″hello″)` *generate different sequences of events.*

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

$$\begin{aligned}
&f_1(\texttt{E}, \gamma :: \Sigma) = ((\texttt{E[TopLevel.main(C.m[\,])]}, \\
&\quad [\gamma(\texttt{attack})/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: [\texttt{new TopLevel()}/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: \gamma :: \Sigma), \epsilon) \\
&f_2(\texttt{E[TopLevel.main(C.m[\,])]}, \gamma'' :: \gamma' :: \gamma :: \Sigma) = ((\texttt{E}, \gamma :: \Sigma), \mathit{iev}(\gamma(\texttt{attack}))) \\
&f_2 * f_1([\,], [\mathbf{a}/\texttt{attack}] :: \varnothing) = (([\,], [\mathbf{a}/\texttt{attack}] :: \varnothing), \mathit{iev}(\mathbf{a}))
\end{aligned}$$

$$p_e(\mathbf{s}_0, f_2 * f_1) = \{\mathbf{s}_0\}$$

$$([\,], [\texttt{new C("hello")}/\texttt{attack}] :: \varnothing) =_{\mathbb{T}} ([\,], [\texttt{new D("hello")}/\texttt{attack}] :: \varnothing)$$

Figure 5.6: Example 5.2.2: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States.

5.3 Taint Analysis Instrumentation via Program Rewriting

In Section 4.3.1, we have defined a rewriting algorithm for both prospective and retrospective enforcement of taint analysis. Since in this section our goal is to study the meaning of prospective taint analysis, we define a more restricted version of $\mathcal{R}$ that only focuses on prospective enforcements through `check` method. In this regard, the target language, $\mathrm{FJ}_{\mathrm{taint}}$, only reflects on prospective checks. The stack-based semantics of $\mathrm{FJ}_{\mathrm{taint}}$ is similar to the semantics of FJ, given in Figure 5.1, as the configurations are defined as pairs of expressions and stacks. One difference is that, in $\mathrm{FJ}_{\mathrm{taint}}$, the integrity event is not generated when a sso is called, but the integrity is generated after `check` passes in the body of that sso. Moreover, the expressions

$\texttt{new TopLevel().main(new C("hello"))} \to_{f_1}^{*}$
$\quad \texttt{TopLevel.main(new B().sso(new C("hello")))} \to_{f_2}^{*} \texttt{new C("hello")}$

$f_1(\texttt{E}, \gamma :: \Sigma) = ((\texttt{E[TopLevel.main(C.m[])]},$
$\quad [\gamma(\texttt{attack})/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: [\texttt{new TopLevel()}/\texttt{this}][\gamma(\texttt{attack})/\texttt{x}] :: \gamma :: \Sigma), \mathit{eev}(\gamma(\texttt{attack})))$
$f_2(\texttt{E[TopLevel.main(C.m[])]}, \gamma'' :: \gamma' :: \gamma :: \Sigma) = ((\texttt{E}, \gamma :: \Sigma), \mathit{iev}(\gamma(\texttt{attack})))$
$f_2 * f_1([\,], [\mathbf{a}/\texttt{attack}] :: \varnothing) = (([\,], [\mathbf{a}/\texttt{attack}] :: \varnothing), [\mathit{eev}(\mathbf{a}), \mathit{iev}(\mathbf{a})])$

$p_e(\mathbf{s}_0, f_1) = p_e(\mathbf{s}_0, f_2 * f_1) = \{\mathbf{s}_0\}$

$([\,], [\texttt{new C("hello")}/\texttt{attack}] :: \varnothing) =_{\mathbb{T}} ([\,], [\texttt{new D("hello")}/\texttt{attack}] :: \varnothing)$

Figure 5.7: Example 5.2.3: The Execution Trace, State Transformers, Attacker Powers and Trust Equivalent Initial States.

are extended with sequences that evaluate according to the following rule:

Sequence

$$(\texttt{v}; \texttt{e}, \Sigma) \to (\texttt{e}, \Sigma).$$

Taint analysis instrumentation adds taint label fields to all objects, and operations for appropriately propagating taint along direct flow paths. We incorporate blocking behavior to enforce blocking checks at secure sinks. In the next section, we will show that this analysis satisfies explicit integrity modulo endorsement, assuming that the library methods are not undertainting.

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

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

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

Figure 5.8: Axioms for Rewriting Algorithm Restricted to Prospective Measures

Definition 5.3.1 *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, 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 5.8.*

Note that in Figure 5.8, we do not need to explicitly specify a case for `endorse`() method body, as μ assigns untainted label for the object returned from that method (Definition 4.3.1).

5.4 Enforcement of Explicit Integrity Modulo Endorsement by $\mathcal{R}$

To satisfy explicit integrity modulo endorsement, $\mathcal{R}$ is required to be an operationally correct implementation of integrity taint policy (Definition 4.4.3). The detailed proofs of operational correctness of $\mathcal{R}$ and semantics preservation have already been studied for substitution-based semantics of FJ in Section 4.4. We avoid to study the same steps for stack-based semantics here due to the obvious bisimiliarty that exists between the two proposed operational semantics. Bisimiliarity states that each step of execution in one semantics can be simulated in zero or more steps in the couterpart operational semantics.

In what follows, we show that the rewritten program satisfies explicit integrity modulo endorsement if the sanity conditions are met. In particular, the library methods must be non-undertainting. In this regard, Lemma 5.4.1 conveys the core property to meet such goal.

Lemma 5.4.1 *Let* $(\mathtt{e}_1, \Sigma_1) \to (\mathtt{e}_1', \Sigma_1')$*,* $(\mathtt{e}_2, \Sigma_2) \to (\mathtt{e}_2', \Sigma_2')$*,* $\mathtt{e}_1, \Sigma_1 =_\circ \mathtt{e}_2, \Sigma_2$ *and* $\mathtt{e}_1 = \mathtt{E}[\hat{\mathtt{e}}]$ *such that* $\hat{\mathtt{e}}$ *is the redex. Then* $\mathtt{e}_1', \Sigma_1' =_\circ \mathtt{e}_2', \Sigma_2'$ *provided that* $(\mathtt{e}_1, \Sigma_1) \to (\mathtt{e}_1', \Sigma_1')$ *does not refer to library method computations, i.e., the following conditions must hold:*

- $\hat{\mathtt{e}} \neq \mathtt{new\ C}(t, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}})$ *for some library method* $\mathtt{C.m}$ *and sequences of values* $\bar{\mathtt{v}}$ *and* $\bar{\mathtt{u}}$*, i.e.,* $\hat{\mathtt{e}}$ *is not a library method invocation.*

- $\mathtt{E} \neq \mathtt{E}'[\mathtt{C.m}([\ ])]$ *for some evaluation context* $\mathtt{E}'$ *and library method* $\mathtt{C.m}$*, i.e.,* $\hat{\mathtt{e}}$ *is not a redex within a library method.*

Proof. By induction on the derivation of $(\mathtt{e}_1, \Sigma_1) \to (\mathtt{e}_1', \Sigma_1')$. Here we study one interesting case. Let $\mathtt{e}_1 = \mathtt{new\ C}(t, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}})$, where $\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}$. We then have $\mathtt{new\ C}(t, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}), \Sigma_1 \to \mathtt{C.m}(\mathtt{e}), [\mathtt{new\ C}(t, \bar{\mathtt{v}})/\mathtt{this}][\bar{\mathtt{u}}/\mathtt{x}] :: \Sigma_1$, assuming that $\mathtt{C.m} \notin$ *SSOs*. Note that the taint tag $\bullet$ is only assigned to primitive objects during evaluation. This can be straightforwardly shown by induction on the derivation of $(\mathtt{e}_1, \Sigma_1) \to (\mathtt{e}_1', \Sigma_1')$. Therefore we have $t = \circ$, otherwise $\mathtt{C.m} \in$ *LibMeths* which is a contradiction. If $\mathtt{new\ C}(\circ, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}), \Sigma_1 =_\circ \mathtt{e}_2, \Sigma_2$ and $(\mathtt{e}_2, \Sigma_2) \to (\mathtt{e}_2', \Sigma_2')$, we need to show that $\mathtt{C.m}(\mathtt{e}), [\mathtt{new\ C}(\circ, \bar{\mathtt{v}})/\mathtt{this}][\bar{\mathtt{u}}/\mathtt{x}] :: \Sigma_1 =_\circ \mathtt{e}_2', \Sigma_2'$.

From $\mathtt{new\ C}(\circ, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}), \Sigma_1 =_\circ \mathtt{e}_2, \Sigma_2$ we have $\mathtt{new\ C}(\circ, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}) =_\circ \mathtt{e}_2$ and $\Sigma_1 =_\circ \Sigma_2$. Using $\mathtt{new\ C}(\circ, \bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}}) =_\circ \mathtt{e}_2$, we conclude that $\mathtt{e}_2 = \mathtt{new\ C}(\circ, \bar{\bar{\mathtt{v}}}).\mathtt{m}(\bar{\bar{\mathtt{u}}})$ such that

$\texttt{new C}(\circ,\bar{\texttt{v}}) =_\circ \texttt{new C}(\circ,\bar{\bar{\texttt{v}}})$, $\bar{\texttt{v}} =_\circ \bar{\bar{\texttt{v}}}$, and $\bar{\texttt{u}} =_\circ \bar{\bar{\texttt{u}}}$. Next, we have $\texttt{new C}(\circ,\bar{\bar{\texttt{v}}}).\texttt{m}(\bar{\bar{\texttt{u}}}), \Sigma_2 \to \texttt{C.m}(\texttt{e}), [\texttt{new C}(\circ,\bar{\bar{\texttt{v}}})/\texttt{this}][\bar{\bar{\texttt{u}}}/\texttt{x}] :: \Sigma_2$. Since $\Sigma_1 =_\circ \Sigma_2$, $\bar{\texttt{v}} =_\circ \bar{\bar{\texttt{v}}}$, and $\bar{\texttt{u}} =_\circ \bar{\bar{\texttt{u}}}$ we conclude that $[\texttt{new C}(\circ,\bar{\texttt{v}})/\texttt{this}][\bar{\texttt{u}}/\texttt{x}] :: \Sigma_1 =_\circ [\texttt{new C}(\circ,\bar{\bar{\texttt{v}}})/\texttt{this}][\bar{\bar{\texttt{u}}}/\texttt{x}] :: \Sigma_2$. This implies that $\texttt{C.m}(\texttt{e}), [\texttt{new C}(\circ,\bar{\texttt{v}})/\texttt{this}][\bar{\texttt{u}}/\texttt{x}] :: \Sigma_1 =_\circ \texttt{C.m}(\texttt{e}), [\texttt{new C}(\circ,\bar{\bar{\texttt{v}}})/\texttt{this}][\bar{\bar{\texttt{u}}}/\texttt{x}] :: \Sigma_2$. □

Lemma 5.4.2 *Let* $(\texttt{e}_1, \Sigma_1) \to^* (\texttt{e}_1', \Sigma_1')$, $(\texttt{e}_2, \Sigma_2) \to^* (\texttt{e}_2', \Sigma_2')$, $\texttt{e}_1, \Sigma_1 =_\circ \texttt{e}_2, \Sigma_2$ *and* $\texttt{e}_1 = \texttt{E}[\hat{\texttt{e}}]$ *such that* $\hat{\texttt{e}}$ *is the redex. Then* $\texttt{e}_1', \Sigma_1' =_\circ \texttt{e}_2', \Sigma_2'$ *provided that* $(\texttt{e}_1, \Sigma_1) \to^* (\texttt{e}_1', \Sigma_1')$ *does not refer to library method computations, i.e., the following conditions must hold:*

- $\hat{\texttt{e}} \neq \texttt{new C}(t,\bar{\texttt{v}}).\texttt{m}(\bar{\texttt{u}})$ *for some library method* $\texttt{C.m}$ *and sequences of values* $\bar{\texttt{v}}$ *and* $\bar{\texttt{u}}$*, i.e.,* $\hat{\texttt{e}}$ *is not a library method invocation.*
- $\texttt{E} \neq \texttt{E}'[\texttt{C.m}([\ ])]$ *for some evaluation context* $\texttt{E}'$ *and library method* $\texttt{C.m}$*, i.e.,* $\hat{\texttt{e}}$ *is not a redex within a library method.*

Proof. By induction on the derivation of $(\texttt{e}_1, \Sigma_1) \to^* (\texttt{e}_1', \Sigma_1')$ using Lemma 5.4.1. □

In the following, let the state transformer defined over a trace τ in FJ_{taint} be denoted by $st(\tau, f)$.

Theorem 5.4.1 *If there does not exist any undertainting library method, then for any arbitrary program* $\mathfrak{p}$*,* $\mathcal{R}(\mathfrak{p})$ *satisfies explicit integrity modulo endorsement.*

Proof. Let $\mathcal{R}(\mathfrak{p}(\mathbf{a})) \Downarrow \sigma\sigma'$. Let $st(\sigma, f)$ and $st(tail(\sigma)\sigma', g)$. Then, according to the definition of state transformers, the state transformer defined over $\sigma\sigma'$ is $g * f$, i.e., $st(\sigma\sigma', g * f)$.

Let the state at $tail(\sigma)$ be $\mathbf{s}$. In order to prove that $\mathcal{R}(\mathfrak{p})$ satisfies explicit integrity modulo endorsement, we need to show that if $\pi_2(g(\mathbf{s})) \notin \mathbf{E_{en}}^*$, then $\pi_2(f(\mathbf{s}_0)) =$

$\pi_2(f(\mathbf{s}'))$ implies that $\pi_2((g * f)(\mathbf{s}_0)) = \pi_2((g * f)(\mathbf{s}'))$ for all state $\mathbf{s}'$ where $\mathbf{s}' =_{\mathbb{T}} \mathbf{s}_0$. This is accomplished by induction on the length of σ'.

When σ' has length 0, $\pi_2(g(\mathbf{s})) = \epsilon$ and so the result is immediate. For the inductive phase, we show that the property holds for σ' of length $n+1$, assuming that for n-length σ' the property is met. So, let's assume the state transformer corresponding to n-length σ' is g_1 and the $(n+1)$'s step defines state transformer g_2. By induction hypothesis we have $\pi_2((g_1 * f)(\mathbf{s}_0)) = \pi_2((g_1 * f)(\mathbf{s}'))$. The goal is to show that $\pi_2((g_2 * g_1 * f)(\mathbf{s}_0)) = \pi_2((g_2 * g_1 * f)(\mathbf{s}'))$. The only interesting case for the last step of execution is where a sso is invoked[1]. So, let the $(n-1)$th configuration of σ' be $(\mathtt{E}[\mathtt{new}\ \mathtt{C}(t, \bar{\mathtt{v}}).\mathtt{m}(\mathtt{new}\ \mathtt{D}(t', \bar{\mathtt{u}}))], \Sigma_{n-1})$ where $\mathtt{C.m} \in \mathit{SSOs}$, and thus nth configuration of σ' be $(\mathtt{E}[\mathtt{C.m}(\mathtt{new}\ \mathtt{C}(t, \bar{\mathtt{v}}).\mathtt{check}(\mathtt{new}\ \mathtt{D}(t', \bar{\mathtt{u}})); \mathtt{e})], [\mathtt{new}\ \mathtt{C}(t, \bar{\mathtt{v}})/\mathtt{this}][\mathtt{new}\ \mathtt{D}(t', \bar{\mathtt{u}})/\mathtt{x}] :: \Sigma_{n-1})$, considering $\mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = \mathtt{x}, \mathtt{e}$.

Note that if $t' = \bullet$, then `check` halts the execution, and no integrity event is generated. Therefore, in this case the result is immediate.

Let $t' = \circ$. Then, `check` does not halt the execution, and the integrity event $\mathit{iev}(\mathtt{new}\ \mathtt{D}(t', \bar{\mathtt{u}}))$ is generated by g_2. By Lemma 5.4.2, since $\mathcal{R}(\mathfrak{p}(\mathbf{a})), [\mathbf{a}/\mathtt{attack}] :: \varnothing =_\circ \mathcal{R}(\mathfrak{p}(\mathbf{a}')), [\mathbf{a}'/\mathtt{attack}] :: \varnothing$ for all attacks $\mathbf{a}'$, the invocations to `C.check` are trust equivalent. Since the same objects are fed to `C.check` and the sso `C.m`, and *SSOs* are only applied on primitive objects, if two such untainted objects are trust equivalent, then they are syntactically equal. Thus untainted object being fed to `C.check` under attacks $\mathbf{a}$ and $\mathbf{a}'$ is of the form $\mathtt{new}\ \mathtt{D}(\circ, \bar{\mathtt{u}})$. This completes the proof.

$\square$

[1]In the rest of the cases integrity events are not generated and so the result is immediate.

5.4.1 Discussion

In this section, we exemplify how taint propagation instrumentation of library methods (in light of Definition 5.2.2 and Definition 5.2.3) affects the satisfaction of explicit integrity modulo endorsement.

To study the role of undertainting library methods assume we have defined taint propagation policy $\text{Prop}(\circ, \texttt{String.concat}(t_1, t_2))$ for library method `String.concat`. Let

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

Consider two attacks $\texttt{new String}(''\texttt{Hello}'')$ and $\texttt{new String}(''\texttt{Attack}'')$. Note that when `concat` is invoked for these two attacks, expressions are trust equivalent, given as the first trust equivalence formula in Figure 5.9. But when library method returns value the expressions are not trust equivalent (second formula in Figure 5.9), since the primitive values are syntactically different but untainted in those expressions. Then explicit integrity modulo endorsement is not satisfied, since attacks generate different events that refine attacker power: $\mathit{iev}(\texttt{new String}(\circ, ''\texttt{Helloworld}''))$ and $\mathit{iev}(\texttt{new String}(\circ, ''\texttt{Attackworld}''))$.

However, if the propagation policy for `String.concat` is defined as (4.1) in Section 4.2.2, then `String.concat` is not undertainting, since the expressions are trust equivalent after return. This is due to the fact that syntactically different primitive values are marked as tainted in those expressions (third formula in Figure 5.9). In this case, both `sso` invocations are halted by `String.check` before generating integrity events, and therefore explicit integrity modulo endorsement is satisfied trivially.

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

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

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

Figure 5.9: Undertainting Example.

To study effects of overtainting, consider primitive operation $prim(str)$ and its OO wrapper `String.primwrapper` with the taint propagation policy given in Section 5.2.3, i.e., $\text{Prop}(t, \texttt{String.primwrapper}(t))$. Assuming

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

any two attacks $\texttt{new String}(str)$ and $\texttt{new String}(str')$ end up invoking `String.sso` on $\texttt{new Int}(\bullet, \texttt{0})$ in the image of $\mathcal{R}$, but are blocked by `String.check` in the body of `String.sso` before generating any integrity event. This represents the unnecessary conservativeness of taint propagation policy for library method `String.primwrapper`.

If the taint propagation policy is defined as $\text{Prop}(\circ, \texttt{String.primwrapper}(t))$ (i.e., `String.primwrapper` to be non-overtainting), then given any two arbitrary attacks $\texttt{new String}(str)$ and $\texttt{new String}(str')$, the integrity event $iev(\texttt{new Int}(\circ, \texttt{0}))$ is generated, which does not affect the satisfaction of explicit integrity modulo endorsement by the rewritten program.

Since each primitive operation Op could be defined arbitrarily and the taint propagation policy predicated on Prop for the library method that wraps Op is user-defined, satisfaction of explicit integrity modulo endorsement depends on the accuracy of the user-defined taint propagation policy for each given primitive operation. Therefore, the framework that has been proposed in this chapter, provides guidelines to implement semantically sound taint propagation mechanisms that enforce direct integrity flow policies in the presence of input sanitization.

Chapter 6

Conclusion and Future Work

The major contributions of this dissertation are as follows:

- The first semantic notion for retrospective enforcement that enables discussing sound (Definition 2.5.2) and complete (Definition 2.5.3) audit logs.
- A rewriting algorithm to enforce retrospective measures (Definition 3.2.3), and prove its soundness and completeness (Theorem 3.2.3).
- The first logical assertion of taint analysis that supports uniform specification and enforcement of the in-depth dynamic integrity taint analysis policies (Definition 4.2.3).
- The first semantic framework for dynamic integrity taint analysis that is inclusive enough to support functional languages along with imperative ones (Definition 5.1.3).

In this dissertation, we establish a formal framework to specify and enforce prospective and retrospective security measures in concert, called in-depth enforcement of security. In this regard, we address the problem of audit log correctness. In particular, we consider how to separate logging specifications from implementations, and how to formally establish that an implementation satisfies a specification. By leveraging the theory of information algebra, we define a semantics of logging specifications. To this end, we formulate information algebras based on first-order logic (Theorem 2.6.1) and show that this formulation enjoys information-algebraic properties.

As a major application space for in-depth enforcement of security, we have provided a general class of logging specifications that employ retrospective security for the sake of surveillance and accountability. We define a particular program rewriting strategy for a core OO calculus that supports instrumentation of this class of specifications expressed in first order logic, and then prove this strategy correct (Theorem 3.2.3). This illustrates how to prove program instrumentation correctness for particular rewriting algorithms leveraging the proposed auditing semantics. This strategy is then applied to develop a practical tool for instrumenting logging specifications in OpenMRS, a popular medical records system. We discuss implementation features of this tool, including optimizations to minimize memory overhead.

Aiming to leverage in-depth enforcement for ameliorating potentially flawed prospective mechanisms, we consider integrity taint analysis in the OO 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. We prove that the proposed rewriting algorithm is operationally correct for the prospective enforcement (Theorem 4.4.2), and retrospectively sound and complete (Theorem 4.4.3).

Moreover, we propose a semantic framework to model direct flow of data integrity enforced by integrity taint analysis techniques. In contrast to previous work, this framework is general enough to model direct flows in both imperative and functional settings and could be used to study the prerequisites for correct taint propagation. We use this framework to extract such requirements for our proposed taint analysis solution and prove the satisfaction of proposed semantic property (Theorem 5.4.1). This exemplifies the use of the proposed semantics to establish provable correctness conditions for a rewriting algorithm that instruments integrity taint analysis in the presence of input sanitization.

6.1 Future Work

Our research on provably correct in-depth enforcement of security policies relies on the proposed semantic framework for audit log generation that is constrained to linear processes. This limitation, in practice, restricts the application of our framework to systems where a single process is responsible to generate audit logs. In our case study on a medical records system, for instance, audit logging capability is considered as an extension to the web server program and all preconditions for logging depend on events in the same program execution trace.

For example, breaking the glass event is a precondition to log accesses to particular patient information in break the glass policies. Instrumentation of medical records web server guarantees correct audit logging as long as such events occur in the execution trace of the web server. This eliminates the possibility of distributing authentication and authorization tasks to other nodes of the network. Such restriction encourages us to study the semantics of distributed audit log generation that underlies correct instrumentation of distributed programs for auditing purposes.

Majority of previous work on audit logging in distributed environments focus on audit log analysis (e.g., [87]) and security concerns regarding in transit and/or at rest log information (e.g., [88, 89, 90]). Studies regarding the collection of logs from multiple monitors in distributed intrusion detection systems is such an example (e.g., [91, 92]). But we currently lack any correctness measures for audit logs generated simultaneously within multiple nodes of a distributed system, where log generation could depend on events in potentially other nodes. This represents the lack of an auditing model in distributed systems, by which retrospective measures could be studied in their fulfillment of accountability goals.

The proposed semantic framework needs to provide a mechanism to specify auditing requirements based on concurrent execution traces. Our framework needs to be general enough to encompass different audit log generation and representation approaches as its instances. We have already shown the generality of information theoretic models in this realm [77]. To this end, I am interested to pursue studying the ways in which these models could be employed to interpret audit logs, specify auditing requirements and define correct enforcement of retrospective security in distributed environments. Similar to our current model for linear processes, correctness of log in distributed environments needs to be conditioned on the specifications of auditing requirement through the comparison between the information contained in the log and the information advertised by those specifications.

We also need to study the languages to specify auditing requirements in distributed environments as instances of our general model. As we have discussed in the previous work, Horn clause logic is a proper language for this purpose in linear executions, due to straightforward modeling of execution traces as sets of facts, sufficient expressivity to specify auditing requirements and available logic programming implementations. Therefore, I intend to investigate the employment of Datalog-like languages for the specification of auditing requirements in concurrent executions.

A formal language model can be used to specify and establish correct enforcement of retrospective security in concurrent systems according to the developed framework. This formalism provides a model for developing tools with correctness guarantees. This model needs to at least enjoy the following features.

- Concurrency: In order to reflect distributed environments, the language model needs to support concurrent process executions with inter-process communication capabilities.

- Generality: The language model needs to be sufficiently concise and high level to describe interactions among processes. Therefore, a wide range of distributed systems could be modeled with this formalism.

- Universal timing: We need to be able to specify the ordering of interesting events for the sake of specifying auditing requirements. For example, in break the glass policies access to particular patient information is logged as long as the glass is already broken. In order to implement such specifications, we need to apply a timing mechanism that is shared among all nodes of the network. Each step of concurrent execution of processes updates this universal time.

- Named functions: To specify auditing requirements, a fundamental unit of secure operations is required to be defined. Functions can be considered as abstractions of these fundamental units in many languages and systems. Therefore, the language model needs to support named functions. When a function is invoked the corresponding function body is required to be fetched and executed. This necessitates a codebase for function definitions. Codebases can be modeled in a straightforward way by mappings from function names to closed language terms.

- Optimizations: The formal specification of retrospective security enforcement in concurrent systems needs to address optimizations regarding bandwidth consumption and memory usage for audit logging. High bandwidth consumption is caused where inter-process communication of logging preconditions becomes overwhelming. Storing logging preconditions in local memory may cause memory overhead. In this regard, we need to refrain from communicating and storing "unnecessary" logging preconditions by each process.

Using the formalism with aforementioned features enables us to deploy distributed environments that guarantee the correct generation of audit logs according to the developed semantic framework. Deployment of correct logging capabilities in health care informatics and intrusion detection systems are examples of case studies that could be considered as final stages of this research.

Our work on the correctness of in-depth policy enforcement and extending it to distributed systems as described above are predicated on given sets of prospective and retrospective policies. Traditionally, these sets of policies are defined according to the expert knowledge that realizes the security requirements and mechanics of the systems being studied. As information systems become more complex in today's world, extracting the required policies that dictate the safety and security of data and code becomes more and more difficult. This encourages us to use machine learning techniques to identify unified in-depth security policies in legacy systems based on a history of executions, access denials, security breaches and audit logs. Such a framework facilitates the long-term adaptivity of legacy systems to the situations that may require policies to be modified, and paves the way for correct enforcement of in-depth security policies that are adjustable in the long run, when accompanied with our frameworks for prospective and retrospective security in linear and concurrent systems. Generally, this approach opens up a wide array of opportunities for future security studies with different applications in healthcare, payment systems, security tools, etc.

In addition to aforementioned research opportunities, multiple theses can be defined on the basis of projects that arise from this doctoral research. These projects may refer to

- legacy systems in which security enforcement is needed to be improved through the addition of retrospective measures beside the existing prospective controls.
- legacy systems that require to be hardened against potential vulnerabilities through in-depth analysis of security.

- taint analysis tools whose formal correctness results are required to be explored.

Moreover, as legacy systems depend on different technologies and platforms and for the sake of generality and support for software with inaccessible source code, it is recommended to define projects that study the deployment of our enforcement techniques in lower level languages, e.g., LLVM-IR, Java bytecode, etc.

References

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

[2] Butler W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, 2004.

[3] Dean Povey. Optimistic security: A new access control paradigm. In *NSPW 1999*, pages 40–45, 1999.

[4] Daniel J. Weitzner. Beyond secrecy: New privacy protection strategies for open information spaces. *IEEE Internet Computing*, 11(5):94–96, 2007.

[5] Audit finds employee access to patient files without apparent business or treatment purpose. http://www.cpmc.org/about/press/News2015/phi.html, 2015. Accessed: 2015-01-30.

[6] Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, Ja<mes A. Hendler, and Gerald J. Sussman. Information accountability. *Communications of the ACM*, 51(6):82–87, 2008.

[7] Wen Zhang, You Chen, Thaddeus Cybulski, Daniel Fabbri, Carl A. Gunter, Patrick Lawlor, David M. Liebovitz, and Bradley Malin. Decide now or decide later? Quantifying the tradeoff between prospective and retrospective access decisions. In *CCS 2014*, pages 1182–1192, 2014.

[8] Jason Tyler King, Ben Smith, and Laurie Williams. Modifying without a trace: General audit guidelines are inadequate for open-source electronic health record audit mechanisms. In *IHI 2012*, pages 305–314. ACM, 2012.

[9] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? An empirical study on logging practices in industry. In *ICSE 2014*, pages 24–33, 2014.

[10] Anton Chuvakin. Beautiful log handling. In Andy Oram and John Viega, editors, *Beautiful security: Leading security experts explain how they think*. O'Reilly Media Inc., 2009.

[11] Richard A Kemmerer and Giovanni Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, 2002.

[12] Duncan Cook, Jacky Hartnett, Kevin Manderson, and Joel Scanlan. Catching spam before it arrives: Domain specific dynamic blacklists. In *AusGrid 2006*, pages 193–202. Australian Computer Society, Inc., 2006.

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

[14] Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, 2003.

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

[16] Pam Matthews and Holly Gaebel. Break the glass. In *HIE Topic Series*. Healthcare Information and Management Systems Society, 2009. http://www.himss.org/files/himssorg/content/files/090909breaktheglass.pdf.

[17] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *CSF 2008*, pages 177–191, 2008.

[18] Anupam Datta, Jeremiah Blocki, Nicolas Christin, Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kirli Kaynar, and Arunesh Sinha. Understanding and protecting privacy: Formal semantics and principled audit mechanisms. In *ICISS 2011*, pages 1–27, 2011.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[38] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kirli Kaynar, and Anupam Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *WPES 2010*, pages 73–82, 2010.

[39] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. Privacy policy specification and audit in a fixed-point logic: How to enforce HIPAA, GLBA, and all that. Technical Report CMU-CyLab-10-008, Carnegie Mellon University, April 2010.

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

[41] Benjamin Böck, David Huemer, and A. Min Tjoa. Towards more trustable log files for digital forensics by means of "trusted computing". In *AINA 2010*, pages 1020–1027. IEEE Computer Society, 2010.

[42] Ricardo Corin, Sandro Etalle, J. I. den Hartog, Gabriele Lenzini, and I. Staicu. A logic for auditing accountability in decentralized systems. In *FAST 2004*, pages 187–201, 2004.

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

[44] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Towards a theory of accountability and audit. In *ESORICS 2009*, pages 152–167, 2009.

[45] Sandro Etalle and William H. Winsborough. A posteriori compliance control. In *SACMAT 2007*, pages 11–20, 2007.

[46] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA 2005*, pages 365–383. ACM, 2005.

[47] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA 2005*, pages 345–364, 2005.

[48] Nataliya Guts, Cédric Fournet, and Francesco Zappa Nardelli. Reliable evidence: Auditability by typing. In *ESORICS 2014*, pages 168–183. Springer-Verlag, 2009.

[49] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. *Lecture Notes in Mathematics - Springer Verlag*, pages 316–330, 2000.

[50] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD 2006*, pages 539 – 550, 2006.

[51] James Cheney. A formal framework for provenance security. In *CSF 2011*, pages 281–293, 2011.

[52] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV data model. http://www.w3.org/TR/2013/REC-prov-dm-20130430, 2013. Accessed: 2015-02-07.

[53] James Cheney. Semantics of the PROV data model. http://www.w3.org/TR/2013/NOTE-prov-sem-20130430, 2013. Accessed: 2015-02-07.

[54] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake Picasso: Preventing history forgery with secures provenance. In *FAST 2009*, pages 1–14, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[70] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Foundations for auditing assurance. In *Layered Assurance Workshop (LAW)*, 2015.

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

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

[73] Debmalya Biswas and Valtteri Niemi. Transforming privacy policies to auditing specifications. In *HASE 2011*, pages 368–375, 2011.

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

[75] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *ICML 2006*, pages 1105–1112. ACM, 2006.

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

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

[78] Syed Zain Rizvi, Philip W. L. Fong, Jason Crampton, and James Sellwood. Relationship-based access control for an open-source medical records system. In *SACMAT 2015*, pages 113–124, 2015.

[79] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Coq formalization of auditing correctness for core functional calculus. `https://github.com/sepehram/auditing-instrumentation-correctness`, 2015.

[80] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[81] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Retrospective Security Module for OpenMRS. `https://github.com/sepehram/retro-security-openmrs`, 2015.

[82] Spring framework. `http://projects.spring.io/spring-framework/`, 2015. Accessed: 2015-09-27.

[83] XSB. `http://xsb.sourceforge.net/`, 2012. Accessed: 2015-09-27.

[84] Logic for your app. `http://interprolog.com/`, 2014. Accessed: 2015-09-27.

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

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

[87] Abdelaziz Mounji, Baudouin Le Charlier, D. Zampuniéris, and Naji Habra. Distributed audit trail analysis. In *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*, pages 102–113, 1995.

[88] Adam J. Lee, Parisa Tabriz, and Nikita Borisov. A privacy-preserving interdomain audit framework. In *Proceedings of the 2006 ACM Workshop on Privacy in the Electronic Society, WPES 2006, Alexandria, VA, USA, October 30, 2006*, pages 99–108, 2006.

[89] Attila Altay Yavuz and Peng Ning. BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 219–228, 2009.

[90] Rafael Accorsi. Bbox: A distributed secure log architecture. In *Public Key Infrastructures, Services and Applications - 7th European Workshop, EuroPKI 2010, Athens, Greece, September 23-24, 2010. Revised Selected Papers*, pages 109–124, 2010.

[91] Yu-Sung Wu, Bingrui Foo, Yongguo Mei, and Saurabh Bagchi. Collaborative intrusion detection system (CIDS): A framework for accurate and efficient IDS. In *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, pages 234–244, 2003.

[92] Vinod Yegneswaran, Paul Barford, and Somesh Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, 2004.