STATIC EXTRACTION OF DATAFLOW COMMUNICATION FOR SECURITY

by

LAURENŢIU RADU VANCIU

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2014

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

To the memory of my beloved grandmother, Maria,

who did not have the chance to go to school

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Marwan Abi-Antoun, for believing in me, and for his guidance and support to fulfill my goal. I am thankful for his patience to keep a close eye on me and to correct my mistakes, and for giving me constant and indispensable feedback.

Then, I would like to thank my dissertation committee members for their valuable feedback and suggestions: Dr. Vaclav Rajlich for sharing his research experience and coaching me how to teach software engineering classes, Dr. Andrian Marcus for opening my eyes to alternatives, and Dr. Jonathan Aldrich for asking me challenging questions and for carefully going over the formal details of my work.

I am also grateful to my teaching advisors, Ms. Monika Witoslawski who recommended me for the teaching award, and to Dr. Nathan Fisher for writing and sending recommendation letters for me.

I would like to thank all the past and current members of the SoftwarE Visualization and Evolution REsearch (SEVERE) group for their support during my journey. Particularly, I thank Maksym Petrenko and Leon Wilson for their support and patience during my first years as a graduate student, and Laura Moreno and Oscar Chaparro for their valuable feedback on my presentations. I would like to thank Suhartha Chowdhury, Chris Dorman, and Andrew Giang who were students in the class I was a teaching assistant, but then they became my colleagues and my friends. I am grateful to Ebrahim Khalaj and Sumukhi Chandrashekar for collaborating with me and for their patience while I was writing my dissertation.

I would like to thank Calin Voichita, Sonia Haiduc, Bogdan Dit, and Claudia Iacob who joined me on this journey. We started our graduate studies by sharing an empty apartment, then each of us found another way towards the Ph.D., and in the end, we all did it! In addition, I would like to thank Calin for asking me hard questions about my research and for his moral support until I found the answers. Many thanks to all my new friends, Michele, Grace, and Elena, who I found during graduate school, and my old friends Paul and Daniel

who still find time to meet me whenever possible.

I would like to thank my parents Stefan and Terezia for their sacrifice and encouragements, although that meant spending most of my time away from them. Many thanks to my sister Cami who initiated me in programming, to my sister Alina who took care of me, and to my brother Sebi for challenging me to push my limits. I would like to thank my brothers and sister in law for their moral support, and my adorable nephews and nieces who learned how to write on Skype while I was writing my dissertation.

Last and foremost, I would like to thank my wife Cristina for her unconditional love and for her constant complaints, which made me want to graduate. I thank her for proofreading my work, which was a necessary but insufficient condition to get the papers published.

Funding. This work was supported in part by Wayne State University, and in part by the Army Research Office under Award No. W911NF-09-1-0273. The views and conclusions contained herein are my own and should not be interpreted as representing the policies or opinions of Wayne State University or any of the sponsors of this research.

TABLE OF CONTENTS

Dedication ii

Acknowledgments iii

List of Tables x

List of Figures xi

Chapter 1: Introduction 1

1.1 Finding Architectural Flaws is Challenging 2

1.2 Architectural Risk Analysis 3

1.2.1 Limitations of Architectural Risk Analysis 4

1.2.2 Object Graph Extraction 4

1.2.3 Security Constraints and Properties 6

1.3 The Scoria Approach 8

1.4 Security Analysis Requirements 9

1.5 Thesis Statement 9

1.5.1 Hypotheses 9

1.5.2 Contributions to Research and Practice 11

1.6 Outline 12

Chapter 2: Extraction of Object Graph 13

2.1 Extracting a Runtime Architecture 14

2.1.1 Running Example 14

2.1.2 As-Designed Runtime Architecture 15

2.1.3 Code vs. Runtime Structure 17

2.2 Dataflow Edges that Refer to Objects 21

2.3 The Extracted Object Graph is Sound 23

2.4 ScoriaX Considers Aliasing 25

2.4.1 Using Type Systems 26

2.4.2 Using a Separate May-Alias Analysis 27

2.4.3 ScoriaX is Domain-Sensitive 29

2.5 ScoriaX Supports Legacy Code . . . 31
2.5.1 The Extracted Object Graph has Traceability to Code . . . 32
2.6 The Extracted Object Graph is Hierarchical . . . 34
2.6.1 Flat Object Graph . . . 35
2.6.2 Object Hierarchy with Ownership Domains . . . 36
2.7 ScoriaX Supports Extensions of Ownership Domains . . . 40
2.7.1 Extensions of Ownership Domains . . . 41
2.7.2 Other Flow Objects . . . 42
2.7.3 Value Flow Graph . . . 42
2.7.4 ScoriaX Algorithm . . . 44
2.7.5 Value Flow Analysis is Domain-Sensitive . . . 45
2.8 ScoriaX Summarizes Runtime Object Graphs . . . 48
2.8.1 Recursive Types . . . 48
2.9 Extracted Object Graph is Precise . . . 50
2.9.1 Architects can Fine-Tune Precision with Annotations . . . 51
2.9.2 ScoriaX is Flow-Insensitive . . . 51
2.9.3 Imprecision of ScoriaX . . . 53
2.10 Scalability of ScoriaX . . . 55
2.11 Summary . . . 56
Chapter 3: Formalization of Extraction . . . 57
3.1 Formalization on Ownership Domains . . . 57
3.1.1 Abstract Syntax . . . 58
3.1.2 Object Graph . . . 58
3.1.3 Static Semantics . . . 61
3.1.4 Dynamic Semantics . . . 67
3.2 Soundness . . . 70
3.3 Extensions of Ownership Domains . . . 75

3.3.1 Extended Syntax . . . 75
3.3.2 Extended Object Graph . . . 76
3.3.3 Extended Static Semantics . . . 77
3.3.4 Flow Graph Analysis . . . 81
3.4 Implementation . . . 86
3.4.1 Implementation Supports Full Java Language . . . 90
3.4.2 Extraction of Other Types of Edges . . . 90
3.5 Previous Versions of ScoriaX . . . 92
3.6 Summary . . . 93
Chapter 4: Evaluation of Extraction . . . 94
4.1 Tool Support . . . 94
4.2 Extraction Methodology . . . 95
4.3 Subject systems . . . 97
4.4 Results . . . 98
4.4.1 ScoriaX Meets the Extraction Requirements . . . 99
4.5 Summary . . . 103
Chapter 5: The Scoria Approach . . . 104
5.1 Scoria Overview . . . 104
5.1.1 The Scoria Process . . . 107
5.1.2 Incrementality of Scoria . . . 108
5.2 Definitions . . . 108
5.3 Security Graph and Queries . . . 110
5.3.1 Data Types . . . 111
5.3.2 Selection Queries . . . 112
5.3.3 Conditions . . . 114
5.3.4 Property Queries . . . 115
5.3.5 Extensibility . . . 115

5.4 Summary . . . 116
Chapter 6: Finding Architectural Flaws . . . 117
6.1 Evaluation on CERT Rules . . . 117
6.1.1 Constraints that Implement CERT Rules . . . 118
6.1.2 Results . . . 125
6.2 Evaluation on Open-Source Application . . . 127
6.2.1 Collecting Security Constraints from Documentation . . . 128
6.2.2 Information Disclosure in UPMA . . . 131
6.2.3 Lessons Learned . . . 132
6.3 Evaluation on Benchmark . . . 135
6.3.1 Quantitative Results . . . 137
6.3.2 Qualitative Analysis . . . 139
6.4 Threats to Validity . . . 146
6.5 Summary . . . 148
Chapter 7: Related Work . . . 149
7.1 Extracting Object Graphs . . . 150
7.1.1 Static Analyses . . . 151
7.1.2 Framework-Based Static Analyses . . . 152
7.1.3 Dynamic Analyses . . . 155
7.2 Finding Security Vulnerabilities . . . 158
7.2.1 AST-Based Static Analyses . . . 158
7.2.2 Android-Specific Analyses . . . 159
7.2.3 Static Taint Analyses . . . 159
7.2.4 Dynamic Taint Analyses . . . 160
7.2.5 Commercial Security Tools . . . 161
7.2.6 Type-Based Approaches . . . 163
7.2.7 Querying Object Graphs . . . 166

7.2.8 Operating System-level Approaches . . . 167
7.2.9 Architectural-Level Approaches . . . 167
7.3 Evaluation of Security Approaches . . . 170
7.3.1 Security Benchmarks . . . 170
7.3.2 Applications with Injected Vulnerabilities . . . 170
7.3.3 Case Studies on Real-World Applications . . . 171
7.4 Summary . . . 172
Chapter 8: Conclusions and Future Work . . . 173
8.1 Limitations . . . 173
8.1.1 Limitations of the Extraction . . . 174
8.1.2 Effort of Writing Constraints . . . 175
8.2 Future Work . . . 176
8.2.1 Improve Extraction . . . 176
8.2.2 Extract Other Types of Edges . . . 176
8.2.3 Field Study on Security Architects . . . 177
8.2.4 Abstract Object Graphs in Software Maintenance . . . 177
8.3 Conclusion . . . 177
Appendix . . . 179
9.1 Theorem: Dataflow Preservation (Subject Reduction) . . . 179
9.2 Theorem: Dataflow Progress . . . 195
9.3 Theorem: Object Graph Soundness . . . 208
9.4 Lemmas . . . 209
Bibliography . . . 213
Abstract . . . 226
Autobiographical Statement . . . 228

LIST OF TABLES

Table 4.1: The process of extracting object graphs and existing tool support . . . 97
Table 4.2: Subject systems . . . 98
Table 4.3: Size of the extracted object graphs . . . 99
Table 4.4: UPMA: Estimated effort to annotate and reason about security. . . . 103
Table 6.1: Summary of constraints that implement five CERT rules . . . 127
Table 6.2: Selected test cases from DroidBench . . . 137
Table 6.3: Recall and precision based on DroidBench . . . 137
Table 6.4: Comparing approaches that find security vulnerabilities . . . 138
Table 7.1: Comparison of static approaches for finding vulnerabilities . . . 150
Table 7.2: Automated detection of the CERT rules . . . 163

LIST OF FIGURES

Figure 1.1: Reasoning about object provenance and indirect communication . . 7

Figure 2.1: CryptoApp: Code fragments. 14

Figure 2.2: As-designed runtime architecture 16

Figure 2.3: Class diagram of CryptoApp extracted using ObjectAid [95]. 18

Figure 2.4: CryptoApp: Code fragments continued from Fig. 2.1. 19

Figure 2.5: Object diagram vs. abstract object graph 20

Figure 2.6: Soundness of abstract object graph 21

Figure 2.7: Dataflow edge refers to object 22

Figure 2.8: Dataflow edges vs. Points-to edges 24

Figure 2.9: Simplified FDJ abstract syntax [10]. 28

Figure 2.10: CryptoApp: Code fragments with ownership types 29

Figure 2.11: Differences between object- type- and domain-sensitivity 30

Figure 2.12: CryptoApp: Code fragments with Ownership Domain annotations. . 33

Figure 2.13: DFD supports hierarchical decomposition 34

Figure 2.14: Indirect communication using objects of type `Socket` 36

Figure 2.15: Data type declarations for the abstract object graph (OGraph). . . . 37

Figure 2.16: Abstract object graph of an object of type `HashMap` 38

Figure 2.17: CryptoApp OGraph visualization 39

Figure 2.18: Three address code version of FDJ syntax 41

Figure 2.19: ScoriaX tracks assignments and extracts the value flow graph (FG) 44

Figure 2.20: The analysis iterates multiple times and stops at a fixed point . . . 45

Figure 2.21: Fragments of value flow graph the analysis uses to resolve `unique` . 46

Figure 2.22: Resolving `lent` and `unique` in the context of object provenance . . . 47

Figure 2.23: Fragments of the value flow graph for the code in Fig. 2.22. 47

Figure 2.24: Handling recursive types, revised from [4, Figure 2.22]. 49

Figure 2.25: Handling recursive types, for a binary tree data structure. 49

Figure 2.26: Placing objects of the same type in different domains 52

Figure 2.27: A flow-insensitive approach reports an information disclosure 53
Figure 2.28: Imprecision due to the invocation of a method in a super class . . . 55
Figure 3.1: Simplified FDJ abstract syntax 59
Figure 3.2: Data type declarations for the OGraph. 60
Figure 3.3: Static semantics. 63
Figure 3.4: Auxiliary judgments for static semantics. 64
Figure 3.5: Static semantics (continued). 66
Figure 3.6: Qualify domains rules. 67
Figure 3.7: Instrumented dynamic semantics (core rules). 68
Figure 3.8: Instrumented dynamic semantics (congruence rules). 69
Figure 3.9: Three-address code version of the FDJ syntax 77
Figure 3.10: Value flow graph data type. For clarity, repeated from Fig. 2.19. . . 77
Figure 3.11: Static semantics of the extraction analysis 78
Figure 3.12: Rules for resolving `lent`, and `unique`. 80
Figure 3.13: Auxiliary judgments for inference rules in Fig. 3.11. 80
Figure 3.14: The algorithm *summarize* 81
Figure 3.15: Working examples for value flow analysis 84
Figure 3.16: Overuse of `lent` and `unique` may lead to false positives 85
Figure 3.17: The algorithm *propagate* 87
Figure 3.18: The analysis iterates multiple times over the code 89
Figure 3.19: Data types to include additional types of edge 91
Figure 3.20: Static semantics for extraction of creation and control flow edges . . 91
Figure 4.1: An example of a root class for an Android application. 96
Figure 5.1: Information content available from communication 105
Figure 5.2: Partial representation of the *SecGraph*. Continued in Fig. 5.3,5.4 . 111
Figure 5.3: Object provenance queries on a *SecGraph* 112
Figure 5.4: Selection and property queries on a *SecGraph* 113

Figure 6.1: Confidential information flows to a descendant of `sckt:Socket` . . . 119
Figure 6.2: Buffer exposed to a malicious client . . . 121
Figure 6.3: Serializing confidential object . . . 123
Figure 6.4: Logging confidential information . . . 124
Figure 6.5: Attacker can execute arbitrary commands . . . 126
Figure 6.6: Commit in UPMA that fixes a security bug . . . 131
Figure 6.7: Constraint: password is sent to object of type `Intent` . . . 131
Figure 6.8: Constraint: password is sent to object of type `ClipboardManager` . 132
Figure 6.9: Code where the password in clear text is disclosed to clipboard . . . 133
Figure 6.10: A fragment of the extracted UPMA object graph . . . 134
Figure 7.1: Listeners code fragments. The complete code is in [130]. . . . 155
Figure 7.2: Sound SDG for the Listeners . . . 156
Figure 7.3: Using tainting type system to find a vulnerability . . . 165
Figure 7.4: Example of a sequence diagram . . . 169
Figure 9.1: Reflexive, transitive closure of the instrumented evaluation relation . 208

Chapter 1 Introduction

With the increased number of mobile devices and applications available, security vulnerabilities at the application-level can lead to the exposure of confidential information such as authentication credentials, location history, and private pictures for a large number of users. To mitigate these threats, a user typically downloads applications available from reputable application stores tied to vendors, and installs up-to-date security software on the device. Vendors use a vetting process to find security vulnerabilities in applications released in their stores [41]. Still, security vulnerabilities continue to be reported in both applications and frameworks [43]. The applications are often developed using object-oriented code; therefore, object-oriented *legacy code* has high business value [57]. The cost of fixing vulnerabilities can be reduced when a security analysis finds vulnerabilities early during the development life cycle, ideally before the application is released.

This thesis proposes a semi-automated approach to support Architectural Risk Analysis (ARA) [86, 121] and finds security vulnerabilities at the architectural level. To achieve its goals, the approach approximates runtime architecture as a sound, hierarchical abstract object graph extracted from code with annotations, where the abstract graph has dataflow edges that refer to objects. To find vulnerabilities, security architects write queries in terms of the hierarchical location, reachability, and provenance of an abstract object. Based on the query results, the architects can reason about the application security and write expressive constraints.

In the rest of this chapter, Section 1.1 discusses why finding vulnerabilities is challenging. Section 1.2 introduces the existing solutions and their limitations. Section 1.3 introduces the proposed approach to support architects in finding vulnerabilities. Section 1.4 lists the requirements the approach meets. Section 1.5 presents the thesis statement, hypotheses and the contributions of this thesis, and Section 1.6 outlines the thesis.

1.1 Finding Architectural Flaws is Challenging

Software defects can be categorized as architectural flaws that are in the design of the application, or coding bugs in the implementation. There is an overlap between the two categories, but categorizing defects is useful because finding defects in each category require a different solution. Finding an architectural flaw may require additional information that is only implicit in the code.

Although 50% of security vulnerabilities are architectural flaws [86] such as information disclosure or misuse of cryptography, they receive less attention from existing static analyses that focus on finding coding bugs such as a hard-coded password or IP address [36]. Finding architectural flaws is difficult because it requires global reasoning about a higher-level representation of the program and not about the code. Finding an architectural flaw such as information disclosure requires reasoning about the context in which confidential information is used. On the other hand, a coding bug can be found by analyzing one class or one method at a time. Approaches focusing on coding bugs can find up to 80% of security vulnerabilities [33]; however, by using only few of the uncovered vulnerabilities, an attacker may compromise the whole system. Moreover, architect still has to manually interpret a reported coding bug and decide if an actual vulnerability exists. For example, an analysis may simply report a vulnerability if an application creates a temporary file that is untrusted, but this is not a vulnerability unless the application stores confidential information such as an IP address or a password into the temporary file.

This thesis proposes an approach (Scoria) that targets architectural flaws that are difficult to find by using existing approaches. Scoria is thus complementary to existing approaches and can be used before the release of a system to analyze the parts that are prone to security vulnerabilities.

1.2 Architectural Risk Analysis

One solution to find architectural flaws is to use Architectural Risk Analysis, also known as Threat Modeling [121]. ARA is one of the three pillars of building secure systems, together with code review and penetration testing [86, Chap. 5]. During ARA, security architects[1] use a forest-level view of the application that allows global reasoning about security from the perspective of an attacker by assigning security properties to component instances and by checking security constraints.

The forest-level view shows runtime components and connectors that reflect how the system works, rather than code entities such as packages and classes that show how the code is organized. Therefore, a code architecture is often insufficient for finding design flaws [86, Chap. 5]. In object-oriented code, a runtime component can be approximated as an abstract object[2] that represents multiple runtime objects of the same type that have the same conceptual purpose. Two runtime components can be of the same type, but serve different conceptual purposes. Similarly, an abstract object graph can have different abstract objects of the same type.

This thesis proposes to use as a forest-level view a runtime architecture that has components and connectors. A connector is more than just a simple relation between components and often shows what information the connector communicates. One goal of this thesis is to approximate the runtime architecture as a graph in which a node represents an abstract object and an edge shows the communicated object.

[1] For brevity, in the rest of this thesis, we refer to a security architect simply as an architect.

[2] This thesis uses *object* with the meaning of an "abstract object". I will use "runtime object" when I mean a location in the heap. The term abstract object is also used in composite terms such as "object identity" to mean "the identity of an abstract object".

1.2.1 Limitations of Architectural Risk Analysis

Currently, ARA is manual and informal with limited support from reverse engineering tools. The tools for extracting runtime architectures are immature [70, 40] compared to the ones for code architecture [91]. Architects draw a diagram representing a runtime architecture manually after interviewing developers and from existing documentation. Since it is difficult for developers to remember all the details, the architecture may miss important components or connectors that exist at runtime. A missed connector such as a dataflow communication may lead to a security vulnerability. For example, a connector that passes confidential information to an untrusted destination may lead to information disclosure. On the other hand, an untrusted connector that has a trusted destination and passes unsanitized information may lead to tampering [121].

1.2.2 Object Graph Extraction

When reasoning about security, ARA considers the worst-case of possible component communication. The analysis results are valid only if they are based on a *sound* architecture that reveals all objects and relations that may exist at runtime, in any program run. Soundness means that there is a mapping such that a unique representative for every runtime object exists, and if a runtime relation exists between two runtime objects, the sound architecture must show an edge between their corresponding representatives. Moreover, if a runtime relation refers to a runtime object, the corresponding abstract relation must refer to the representative of this runtime object. Soundness requires a static analysis to extract the architecture from source code in a way that covers any possible execution, rather than a dynamic analysis, which by definition considers only a finite number of program runs [112].

Soundness also means that the extracted graph has a unique representative for every runtime element. In the code, multiple variables may alias the same runtime object and

the same variable may alias multiple runtime objects. Aliasing enables expressing design that would not be possible otherwise, but at the same time aliasing can make reasoning about security difficult [58]. Soundness can be proven formally and guarantees that no one runtime element has two different representatives in the abstract object graph. Various static analyses were proposed to compute a may-alias relation, and in general, the more precise a may-alias analysis is, the less it scales. Aliasing can also be described using ownership types. In Ownership Domains [10] for example, objects are in named, conceptual groups, and two variables that refer to objects in different groups cannot alias, but the ones that refer to objects in the same domain may alias.

A runtime architecture needs to fit in one page so that it conveys a high-level understanding of the system [86, Chapter 5], and shows how the system can be decomposed into subsystems. Therefore, another requirement of a sound runtime architecture used for security is *hierarchical decomposition*. In plain Java code, a hierarchical organization of objects that conveys architectural abstraction is not directly available.

The solution is provided by Ownership Domains annotations [4, 5] that architects can add to express design intent as local, modular hints about architectural tiers, logical containment and strict encapsulation that is unavailable in plain Java code. Then, a static analysis uses abstract interpretation of code with annotations and extracts a hierarchical object graph, where an abstract object has zero or more domains, where a domain is a named, conceptual group that in turn has one or more objects. The architects can refine the design intent to increase the precision of the extracted graph. However, the existing static analysis [4, 5] extracts only edges representing points-to relations, i.e., a persistent relation based on the field reference between two objects. In addition, an edge needs to track data flow, i.e., a transient relation at runtime that represents dataflow communication between objects. The abstract object graph shows any possible communication that may occur at runtime, and the architects can reason how confidential data may flow between components, and in particular, if confidential data crosses trust boundaries.

Dataflow edges that refer to abstract objects

This thesis proposes an analysis that extracts dataflow communication edges where an edge label is the identifier of an abstract object that can be anywhere in the abstract object graph. In that case, we say that an edge refers to an abstract object that is being communicated. This enables a security constraint to check if an abstract object flows from a source to a destination. I use *dataflow communication* to mean a particular kind of relation between abstract objects due a method invocation, field read, and field write expression. I use *value flow* to mean a *data flow* between variables.

1.2.3 Security Constraints and Properties

Security policies and constraints are only informally described in the documentation of applications, frameworks, or security protocols a framework or a library implements. To check these constraints automatically, architects can formalize the security specification as machine-checkable constraints and find vulnerabilities. Related work includes static taint analyses [48, 49, 126] that track values from sources to sinks, and tainting type systems [37, 92] that allow architects to enhance types with security properties. In these approaches, the constraints are not clearly separated from the analysis or from the type system, which makes their evolution difficult when new vectors of attack are discovered.

An approach such as Scoria that separates security constraints from extraction is extensible. By using dataflow edges that refer to objects, the architects can write machine-checkable constraints in terms of *object provenance* where the same object that a first dataflow edge refers to cannot be referred from a second dataflow edge. Since the abstract object graph is hierarchical, a constraint can be written in terms of descendants and ancestors of abstract objects such that it checks if a dataflow edge with an untrusted destination refers to an ancestor of an object representing confidential data. By using object hierarchy, a constraint can also check *indirect communication*, where confidential data flows to a descendant of untrusted

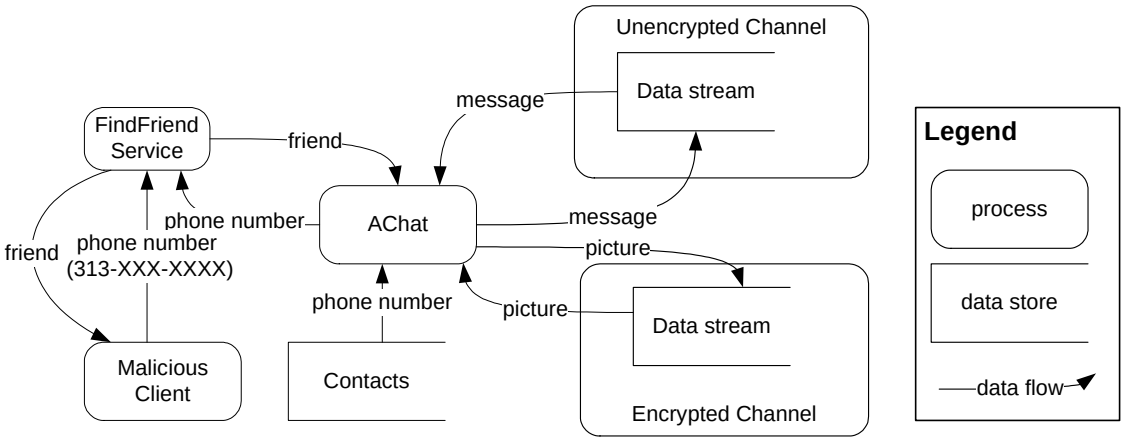


Figure 1.1: A runtime architecture representing a chat application. Object provenance checks that the same abstract object that flows from AChat to the FindFriend Service does not flow from an external client to FindFriend Service. Contact information is disclosed although no transitive information flow exists from Contacts to the External Client. Another constraint uses indirect communication to check that AChat does not send pictures to a data stream nested into the Unencrypted Channel.

object. Architects may also consider other types of communication like points-to edges and creation edges. A constraint can check if the dataflow with an untrusted destination refers to another object from which the confidential data is reachable (*object reachability*).

As an example of an object provenance constraint, consider a chat application (AChat) that allows users to search for their friends that use the same app based on the phone number stored on the device. All the instances of AChat running on different devices use a service available online that is provided by the designers of the app. A constraint that uses object provenance can check that the same abstract object representing the phone number can flow from the app to the service, but cannot flow from an external client to the service. Otherwise, the external client can conduct a brute force attack and try to guess all the phone numbers in an area code to find all the users of AChat in the area, although there is no direct or transitive information flow from the AChat to the external client (Fig. 1.1).

AChat provides two communication channels, one that provides encryption for sending pictures, and one without encryption for instant messaging. The application does not communicate directly to the channel, but rather to a data stream nested in the communication channel. Hence, a constraint needs to use indirect communication and check that pictures are sent only to descendants of a channel that provides encryption.

1.3 The Scoria Approach

Scoria[3] is a semi-automated approach that uses a static analysis called ScoriaX to support the architects to find security vulnerabilities that are caused by architectural flaws. Scoria assumes the source code is available and supports architects to find security vulnerabilities by querying an abstract object graph that the static analysis extracts from the source code. The object graph is sound and hierarchical, and has dataflow edges that refer to objects.

Scoria separates the constraints from the extraction by allowing architects to execute constraints on the results of querying the extracted object graph. The architects can also assign security properties to abstract objects and edges. Scoria is thus extensible and allows architects to adapt to different vectors of attack that the designers of the security analysis might not have considered. Due to first having dataflow edges that refer to objects and second having a unique representative for every runtime object, the architects can write constraints in terms of *object provenance* (check if an object that flows from a source to a destination also flows from another source to another destination) and *indirect communication* (check if confidential data flows to an object nested in an untrusted component).

The thesis includes end-to-end evaluations to show that Scoria can find vulnerabilities in systems by executing machine-checkable constraints. The thesis also compares Scoria with taint analyses based on a benchmark in terms of precision and recall. The results show that by adding annotations and writing constraints Scoria can find vulnerabilities that other approaches miss and avoid false positives. The thesis also extends the benchmark with new test cases that require reasoning about object provenance to find security vulnerabilities and reasoning about object hierarchy and indirect communication to avoid false positives.

Scoria meets the requirements of ARA. The next two sections distinguish requirements of ARA approaches from the requirements of extraction. Throughout the thesis, I explain how Scoria meets these requirements by design or based on the empirical results.

[3]Scoria stands for **S**ecurity **C**onstraints **O**n **R**unt**I**me **A**rchitecture.

1.4 Security Analysis Requirements

SC1: **Use a runtime architecture.** ARA requires a runtime architecture that is consistent with the code and shows components of the same type that have different conceptual purposes, and possibly different security properties. Connectors show all types of communication including dataflow communication that shows "what" data is being communicated from one component to another.

SC2: **Support automation.** Security constraints are often formulated informally in the documentation. An ARA approach needs to provide tool support for the architects to execute machine-checkable constraints to enforce the informal constraints.

SC3: **Extensibility of constraints and properties.** To defend a system against malicious users, an ARA approach should be extensible so that architect can adapt and write constraints separately from the extraction of the runtime architecture. Security design intent may not always be available in the code and architects need flexibility in assigning security properties.

1.5 Thesis Statement

Finding security vulnerabilities that require reasoning about dataflow communication is challenging for architects. A static analysis can extract dataflow communication edges that refer to abstract objects in a sound, hierarchical object graph such that the architects can find vulnerabilities by writing machine-checkable constraints in terms of object provenance and indirect communication through hierarchy and reachability of abstract objects.

1.5.1 Hypotheses

To support the thesis statement, I evaluate two hypotheses:

H1. For an object-oriented program, the architect adds annotations and uses a static analysis to extract dataflow communication edges that refer to objects in a sound, hierarchical object graph that approximates the runtime architecture of the program, and meets the following extraction requirements.

a) *Express design intent:* the annotations express local, modular hints about architectural tiers, logical containment and strict encapsulation; the architect can refine the annotations to increase the precision of the extraction analysis, as needed;

b) *Support legacy code:* the architect adds only annotations to the code and any libraries in use while using existing tools and development environments. The analysis supports extraction without requiring architects to re-engineer the code in a specific programming language or using a specialized library.

c) *Sound object graph:* the architect assumes that the object graph shows all possible objects and edges that may occur in any possible program run;

d) *Sound object graph with respect to aliasing:* the architect does not encounter any situation where the same runtime object is represented by multiple abstract objects in the object graph, so he cannot assign multiple security properties to the same runtime entity;

e) *Precise object graph:* while a sound object graph is inherently going to have false positives, ideally, the architect should not have to frequently investigate too many false positive objects or edges that do not exist in any program run;

f) *Summarization:* By summarizing multiple runtime objects of the same type to a pair (type, domain) corresponding to an abstract object, the analysis allows the architect to express that multiple runtime objects have the same conceptual purpose or the same security property, even if they are created in different locations in the code. Then, the architect does not need to frequently distinguish between runtime objects that have the same corresponding abstract object in the object graph;

g) *High-level view:* the architect visually inspects the hierarchically laid out object graph frequently in order to better understand how to refine the annotations, set the properties,

or refine the constraints;

h) *Scalability:* the extraction analysis needs to scale to real-world applications of at least a few thousands lines of code. Tradeoffs needs to be considered: the more precise the extraction analysis is, the less scalable it is.

H2. Based on a sound, hierarchical, abstract object graph with dataflow edges that refer to objects, the architect writes machine-checkable constraints in terms of one or more of the following Scoria features to effectively (high recall and precision) to detect security vulnerabilities.

a) *Object provenance:* check if the same abstract object that flows from a first source to a first destination also flows from a second source to a second destination;

b) *Object transitivity:* track the flow of potentially shared objects through the system;

c) *Object hierarchy:* identify protected data is nested within abstract objects that ought to carry only unprotected data;

d) *Object reachability:* identify when protected data may be reachable via unexpected communication paths;

e) *Traceability:* by using the traceability between abstract objects and edges and the code, the constraints can use information such as subtyping between classes, method names of interest, or file names, among others.

f) *Indirect communication:* identify when protected data is sent to an abstract object nested within an untrusted sink.

g) *Security property:* enhance the abstract object graph with security design intent that the extraction analysis does not extract from the code;

1.5.2 Contributions to Research and Practice

The thesis makes the following contributions and focuses on extracting the internal representation of the runtime architecture and on finding security vulnerabilities. The thesis deemphasizes the role of visual inspection in finding security vulnerabilities, focusing instead

on security constraints. I assume that the extracted graph can be integrated in existing visualization approaches, and be used for architectural conformance [4], but these contributions are not in the scope of this thesis.

- A more formal and rigorous ARA process that uses an abstract object graph that is consistent with the code instead of manually drawn diagrams;
- Semi-automated support for extracting dataflow edges that refer to abstract objects in a sound, hierarchical object graph that architects can use to reason about security;
- Semi-automated support for reasoning about security by querying the abstract object graph and by writing machine-checkable constraints that make ARA more repeatable and rigorous compared to an informal process;
- Expressive constraints in terms of object provenance and indirect communication that identify when data is trusted or protected based on its origin, and identify when protected data is sent to a component nested within an untrusted sink.

1.6 Outline

The rest of the thesis is organized as follows, where each chapter discusses how Scoria meets the identified requirements. Chapter 2 describes informally the extraction analysis ScoriaX. Chapter 3 formalizes ScoriaX and proves that the analysis is sound for programs written in a Java-like language. Chapter 4 evaluates ScoriaX and discusses precision and scalability. Chapter 5 describes more precisely the Scoria process and the queries the architects can write on the extracted model. Chapter 6 evaluates Scoria by implementing informal constraints, by finding vulnerabilities in an Android application, and by comparing Scoria with other approaches based on a benchmark. Next, Chapter 7 presents related work, and Chapter 8 discusses some limitations of Scoria, presents directions for future work, and concludes.

Chapter 2 Extraction of Object Graph

This chapter[1] describes informally how Scoria uses a static analysis (ScoriaX) to meet the extraction requirements (Section 1.5.1, page 10) and why a runtime architecture is needed. During Architectural Risk Analysis (ARA) [61, 86], security architects find vulnerabilities by analyzing a forest-level view of the program rather than reading code. This forest-level view is a representation of the runtime architecture [124], which is a collection of runtime components and connectors. A runtime architecture can have multiple component instances of the same type that serve different conceptual purposes or have different security properties. A connector has a direction and shows "what" data is communicated from one component to another. To find security vulnerabilities, the architects assign security properties and write constraints. For example to find information disclosure, architects assign confidentiality and trust level to components and connectors and check that no confidential data flows into an untrusted component.

ScoriaX extracts dataflow edges in a sound, hierarchical object graph from object-oriented code with annotations. At runtime, an object-oriented program can be represented as a Runtime Object Graph (ROG). A node in an ROG corresponds to a runtime object and an edge corresponds to a relation between runtime objects such as a points-to or a dataflow communication. The goal of ScoriaX is to extract an abstract object graph that approximates all ROGs in any program run and meet the extraction requirements (Section 1.5.1, page 10). The first section describes what kind of program representations can be used in ARA. Next, Section 2.2 gives an informal definition of dataflow edges that ScoriaX extracts. Then, for each requirement, the rest of the sections describe the solution ScoriaX adopts and the alternatives considered.

[1] An earlier version of this analysis appeared in [131]. The current version appeared in [132] and [133].

```
1  class Main {
2   EActivity eAct = new EActivity();
3  }
4  class EActivity extends Activity {
5    Service service = new Service("ENCRYPT");
6    void onCreate(){
7      FileManager fm = (FileManager)getManager("FILE");
8      LocationManager lm = (LocationManager)getManager("LOCATION");
9      Location loc = lm.getLastLocation();
10     File tempFile = fm.read("history");
11     if (tempFile.find(loc)) {
12       File encrypted = service.encrypt(tempFile);
13       Intent i = new Intent(ACTION_SEND);
14       i.putExtra(EXTRA_STREAM,encrypted);
15       Activity act = new ViewActivity();
16       act.startActivity(i);
17     }
18   }
19 }
20 class ViewActivity extends Activity { ... }
21 class Activity {
22   Intent mIntent;
23   void startActivity(Intent intent) {
24     mIntent = intent;
25   }
26   Manager getManager(String name) {
27     ...//return FileManager or LocationManager
28   }
29 }
30 class Intent{
31   void putExtra(String name, Object value) { ... }
32 }
```

Figure 2.1: CryptoApp: Code fragments.

2.1 Extracting a Runtime Architecture

This section gives an overview of ARA and uses a running example to explain the main concepts. It then describes alternative representations of a runtime architecture and how they can be used in ARA.

2.1.1 Running Example

The running example is an Android application (CryptoApp) that searches in the log history of the device looking for location data. If the current location is found, CryptoApp

encrypts the log file to prevent disclosure of the location to malicious applications. The code fragments of CryptoApp (Fig. 2.1) show the classes `EActivity` and `ViewActivity` that both extend the class `Activity` of the Android framework. The object of type `EActivity` communicates with an object of class `Service` that encrypts the content of a file. The code uses the standard communication mechanism in Android based on objects of type `Intent`.

To access the location information, which is confidential, CryptoApp asks the user to explicitly grant security permissions when the application is installed. The request seems legitimate since the application needs to search for the current location in the files. Once permissions are granted, nothing prevents CryptoApp from saving the location in clear text in a temporary file. The information is disclosed because other applications installed on the device can read temporary files.

Notation. I use the *italic font* for component instances, and the `fixed width font` for code entities and objects. An object labeled `obj:T` indicates a reference `obj` of type `T`, which we then refer to either as the "object `obj`" or the "`T` object" to mean "an instance of the `T` class".

2.1.2 As-Designed Runtime Architecture

ARA requires a runtime architecture, which is a collection of runtime components and connectors, their properties and constraints on their interactions. In a runtime architecture, a component represents runtime—rather than code—entities [86]. One style of runtime architecture used in ARA is a Data Flow Diagram (DFD), which describes how data enters, leaves, and traverses the system (Fig 2.2). It shows data sources and destinations, relevant processes that data goes through, and trust boundaries in the system. A DFD allows architects to find security vulnerabilities without having to search through the source code.

Data-Flow Diagram. A DFD has several component types: a *Process* (circle), a *HighLevelProcess* (two concentric circles), an *ExternalInteractor* (rectangle), or a *DataStore* (two

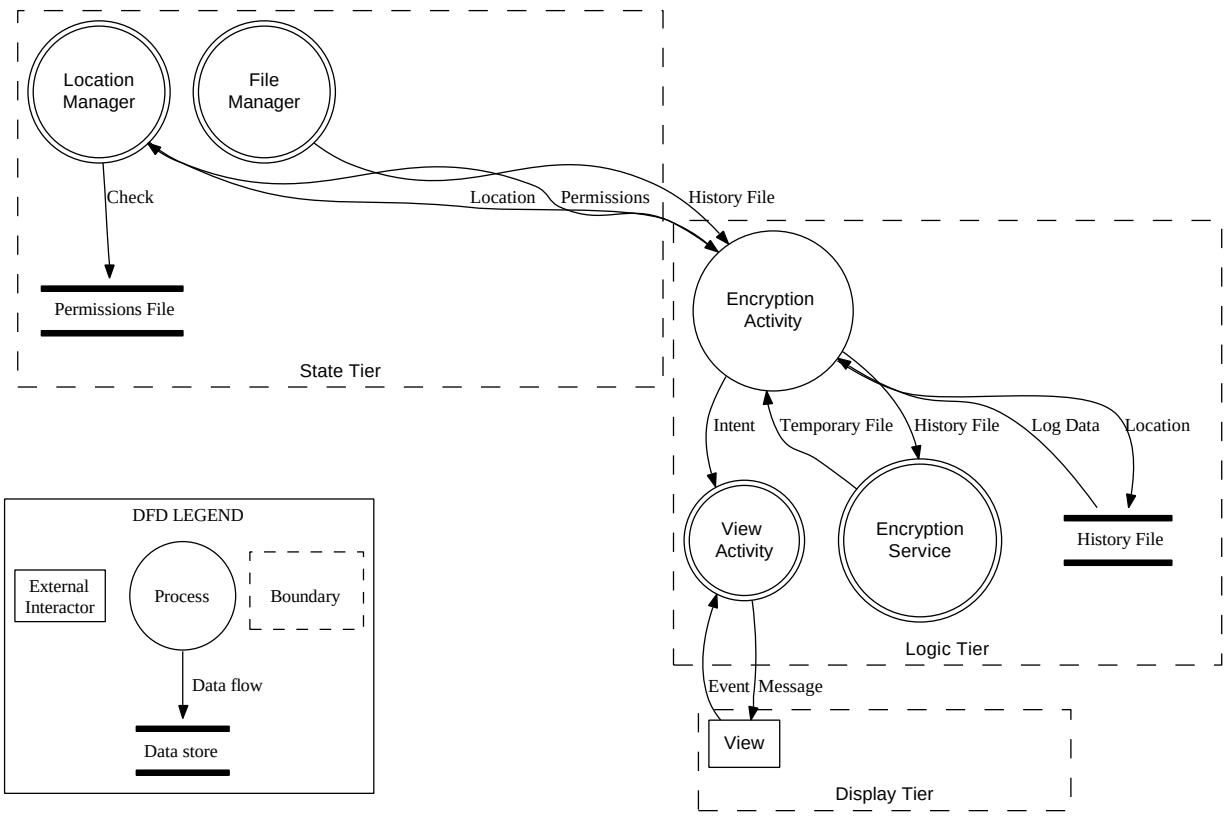


Figure 2.2: As-designed runtime architecture using Data-Flow Diagram style shows an Android application that follows the State-Display-Logic architecture.

parallel horizontal lines). A *Process* is a component that processes data and performs some actions based on the data. A *DataStore* represents a repository such as a file or a database. An *ExternalInteractor* represents a component that exists outside the system such as a web service or a user interacting with the system. The connector *DataFlow* represents data transferred between components where the label describes information content. A DFD used in ARA often separates components that have different privilege levels using a *Boundary* (dashed line). For example, a boundary can describe locations where an attacker can access, or a machine or process boundary [80, p. 90]. A DFD *Boundary* is a *TrustBoundary* (the base type), *ProcessBoundary*, *MachineBoundary* or *OtherBoundary*. Alternatively, *Boundary* can also be a *runtime tier* that is a group of conceptually related component instances. In a DFD, two component instances of the same type can serve different concep-

tual purposes. The DFD shows multiple instances of the same component type such as a *DataStore* representing a file, and the instances can serve different purposes. For example, a file that contains a history log serves a different conceptual purpose than a file that stores permissions granted by the user.

DFD constraints. To find vulnerabilities such as information disclosure, the architects assign security properties such as *TrustLevel* to components and connectors. Then architects write and execute constraints such that no connector passes confidential information from a trusted source to an untrusted destination. For CryptoApp, *Permissions File* is considered trusted while *History file* is considered untrusted. The constraint ensures that no connector passes location information in clear text to an untrusted file.

2.1.3 Code vs. Runtime Structure

A diagram that shows methods, classes and packages represents the code structure and is easier to extract than a diagram that represents the runtime structure. A code structure diagram shows how the code of the program is organized rather than a view of the program at runtime and is unsuitable for ARA. Examples of code structure diagrams are class diagram and call graph, while an example of a runtime structure diagram is the object diagram [123]. Next, I give examples of security constraints that can be written on each diagram and discuss advantages and limitations.

Class diagram. In a class diagram, a node represents a class with field and method signatures (Fig. 2.3). A common relation in a class diagram is inheritance. With inheritance, a class diagram shows the responsibility of an object divided into multiple classes. For example, the method `onCreate` is partially implemented by `EActivity`, while the default responsibilities are implemented by the base class `Activity` (Fig. 2.1). A class diagram also shows only one class but different instances of the same class can serve different conceptual purposes.

Class diagram constraints. If the architects were to use a class diagram to write con-

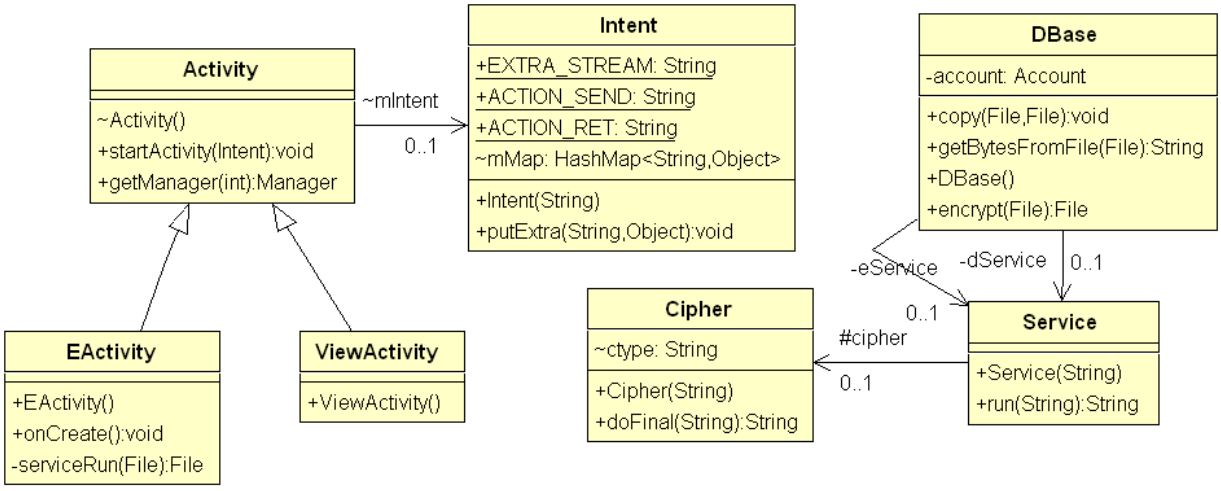


Figure 2.3: Class diagram of CryptoApp extracted using ObjectAid [95].

straints, they would be unable to assign different security properties to different instances of the same class. They may also be misled by multiple classes that correspond to the same instance, since they can assign different security properties to classes in an inheritance hierarchy. In CryptoApp, the client code `DBase` uses two instances of type `Service`, one for encryption and one for decryption (Fig. 2.4). In turn, each instance of `Service` uses its own instance of `Cipher`. While a class diagram shows one box for the class `Service` and one for the class `Cipher`, the architects needs to distinguish between different instances of these classes and identify the instance that decrypts information and returns confidential information in clear text, which is not to be disclosed into an untrusted sink such as a temporary file.

Call graph. In a call graph, a node represents a method declaration and an edge indicates that the source method invokes the destination method. At runtime, the same method can be used in different contexts. Distinguishing between contexts allows architects to avoid false positives. The code fragments in Figure 2.4 show the implementation of the class `Service`, where the method `run` sends the text to an object of type `Cipher`, which transforms the text and sends the result to the client. Depending on the receiver of the method, i.e., the object in the context of which the method is invoked, the method `run` encrypts or decrypts data.

```
1  class DBase {
2    Service eService = new Service("ENCRYPT");
3    Service dService = new Service("DECRYPT");
4    File encrypt(File src) {
5      File dst = new File(src.getName());
6      String mydata = getBytesFromFile(src);
7      String dd = dService.run(mydata);
8      String eData = eService.run(dd);
9      PrintWriter pw = new PrintWriter(dst);
10     pw.println(eData);
11   }
12 }
13 class Service {
14   Cipher cipher;
15   Service(String cType) {
16     cipher = new Cipher(cType);
17   }
18   String run(String text) {
19     return cipher.doFinal(text);
20   }
21 }
22 class Cipher { ... }
```

Figure 2.4: CryptoApp: Code fragments continued from Fig. 2.1.

Call graph constraints. The architects need to ensure that only encrypted data can flow into a temporary file. Assigning security properties may be possible using method declarations such that the architects can select methods that return confidential information (sources), and methods that disclose information (sinks). However, a call graph does not allow architects to write constraints in terms of dataflow communication.

Architects may use other representations such as a System Dependence Graph (SDG), which has both control and data flow edges [126]. The constraint then checks if a SDG contains a path from a source to a sink. Selecting a method declaration as a source or sink can be however imprecise when the method returns both confidential and non-confidential information, as it is the case for the method `run` of class `Service`. Section 7.1.2 (page 152) discusses SDG and other program representations in more detail.

Manually drawn object diagram. One diagram of object-oriented programs that represents runtime structure is an object diagram [123]. An object diagram shows multiple instances of the same class and shows points-to relations between objects. For example, it

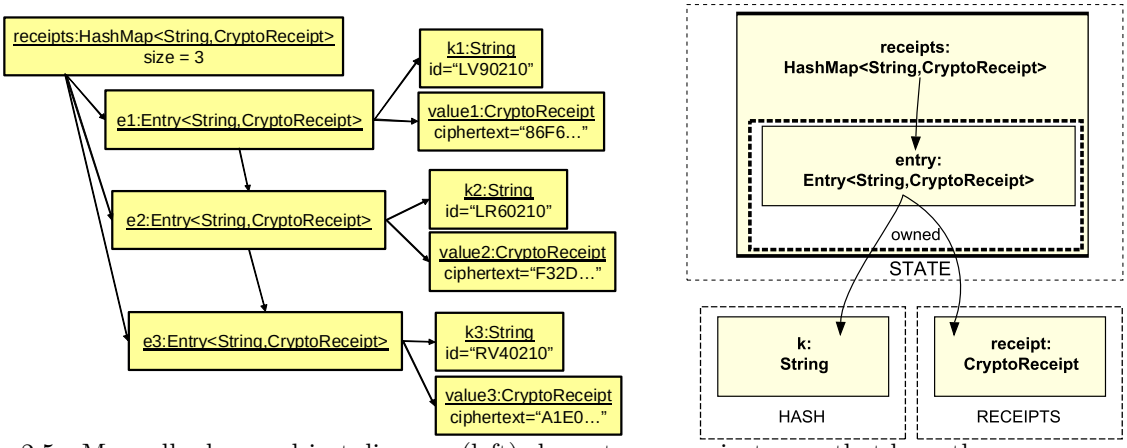


Figure 2.5: Manually drawn object diagram (left) shows too many instances that have the same conceptual purpose. An abstract object graph (right) merges such instances into canonical abstract objects.

shows two instances of `Service` and the field declaration `Cipher c` in the class `Service` corresponds to a points-to relation from `eService:Service` to an object of type `Cipher`. It is not clear only from a field declaration in the code if multiple objects of type `Service` share the same object of type `Cipher`, or if multiple objects of type `Cipher` exist. The object diagram clarifies this situation and shows two objects of type `Cipher`, one for each object of type `Service`. One disadvantage of an object diagram is that it also shows multiple instances of the same class that serve the same conceptual purpose. For an object of type `HashMap`, an object diagram shows for example three objects of type `Entry` that each refers to a key and a value. In the general case, however, the object diagram can have an unbounded number of objects and edges. Consider for example a server that maps a password with an access token. As the server runs indefinitely, it creates an unbounded number of entries in the map.

Object diagram constraints. If architects were to use an object diagram to write constraints, they would be able to distinguish between different instances of the same class, and reason about the presence and the absence of communication. There might be, however, too many instances on such a diagram that serve the same conceptual purpose (Fig. 2.5). Architects would need to apply the same constraint on many similar instances.

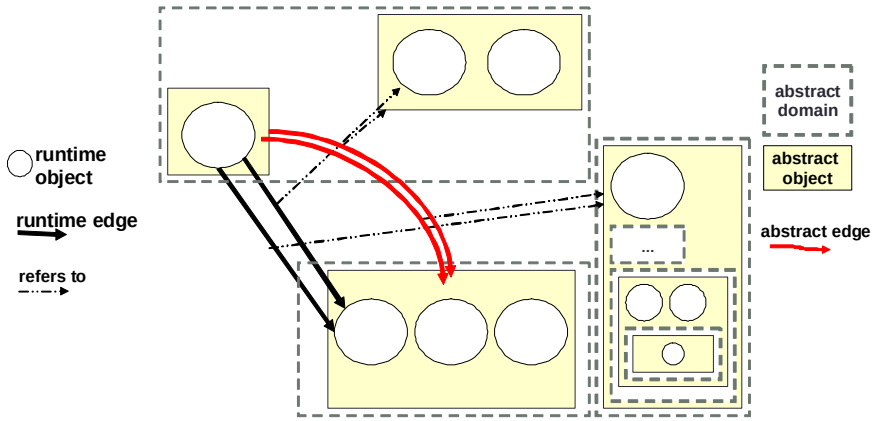


Figure 2.6: An object graph is sound if there is a mapping from any runtime object graph to the abstract graph. If two runtime edges have the same source and destination (runtime objects), but refer to two runtime objects that have distinct representatives in the abstract graph, then the runtime edges have two distinct representative abstract edges in the abstract graph.

2.2 Dataflow Edges that Refer to Objects

A related approach [5, 7] proposes a static analysis that extracts an abstract object graph to approximate a runtime architecture [7]. In an abstract object graph, an abstract object corresponds to multiple runtime objects that have the same conceptual meaning. The abstract objects are organized in groups, where an abstract object can also have groups such that the graph is hierarchical (Fig. 2.6). An abstract object and its substructure corresponds to a *Process* in a DFD. A dataflow edge corresponds to a *DataFlow* connector that refers to a component instance representing the information being sent, such as a *message*, rather than a type or a method invocation [124].

The approach makes a simplistic assumption about dataflow edges, namely that the presence of dataflow communication can be approximated by a points-to edge. Reasoning about security requires more than just reasoning about the absence or the presence of communication. Architects reason about confidential data that is disclosed to an untrusted destination, or tainted data that flows to a trusted destination without being sanitized. Therefore, a connector of the runtime architecture is more than just a simple relation between components. For object-oriented code, a connector corresponds to a dataflow edge for method invocation,

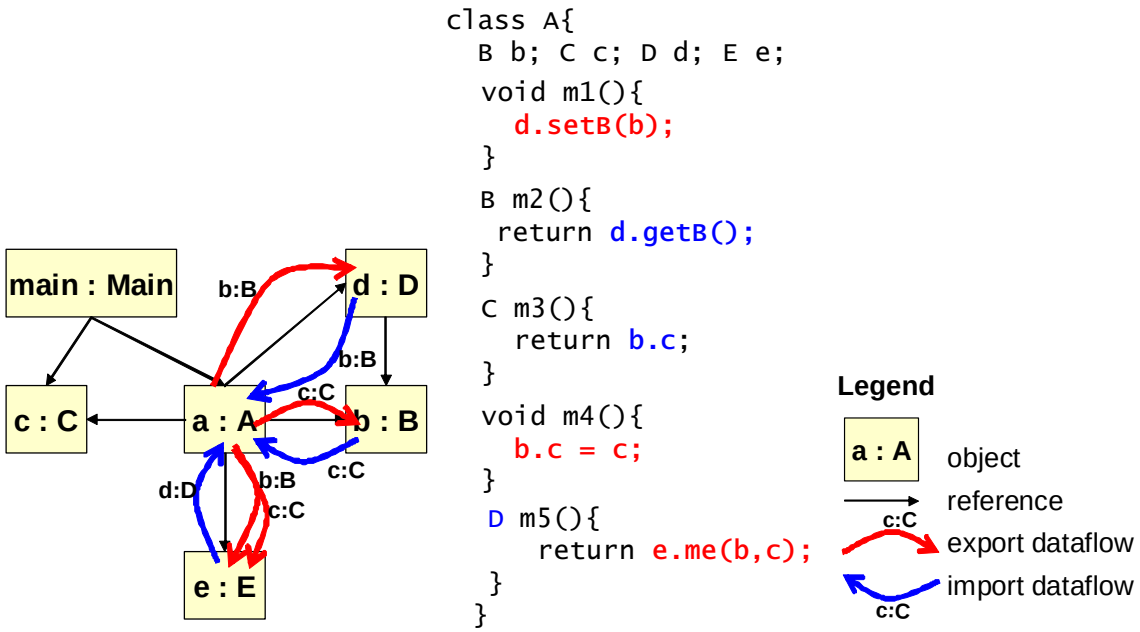


Figure 2.7: In an object graph a node is an abstract object that traces to code to an object allocation expression. An edge represents dataflow communication that refer to objects. Dataflow edges traces to code to field read, field-write and method invocation expressions.

field read, and field write expressions in the code. To show what information is flowing, this thesis enriches the representation of a connector and makes a dataflow edge refers to an object, which can be different from the source and the destination of the dataflow edge (Fig. 2.6).

Definition: Dataflow communication that refers to an object. An object `a:A` has a reference to an object `o:O` and passes it to an object `b:B`, or an object `a:A` has a reference to an object `b:B` and receives a reference to an object `o:O`. The objects `a:A` and `b:B` represent the source or destination objects, and `o:O` is a dataflow object that the dataflow edge refers to. To capture the directionality of the flow, the object graph distinguishes import and export edges. An import dataflow edge exists due to the return value of a method invocation or a field read expression. An export dataflow edge exists due to an argument of a method invocation or a field write expression (Fig. 2.7).

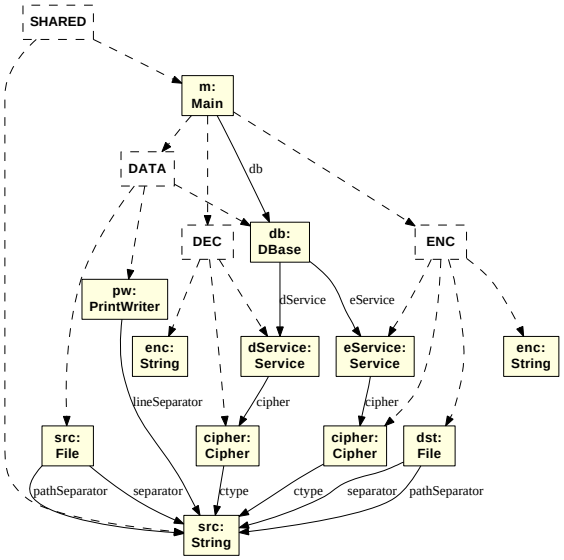
Alternatives considered

If the architects were to use an object graph with points-to edges [5, 7], they would be able to distinguish between objects of the same type, but would miss dataflow communication edges that may lead to security vulnerabilities. In CryptoApp, the architects would distinguish the objects `Service` and `Cipher` that encrypts information. However, the graph shows no points-to edge from `db:DBase` to `pw:PrintWriter` (Fig. 2.8(a)) because `pw` is a local variable, not a field (Fig. 2.4, line 9).

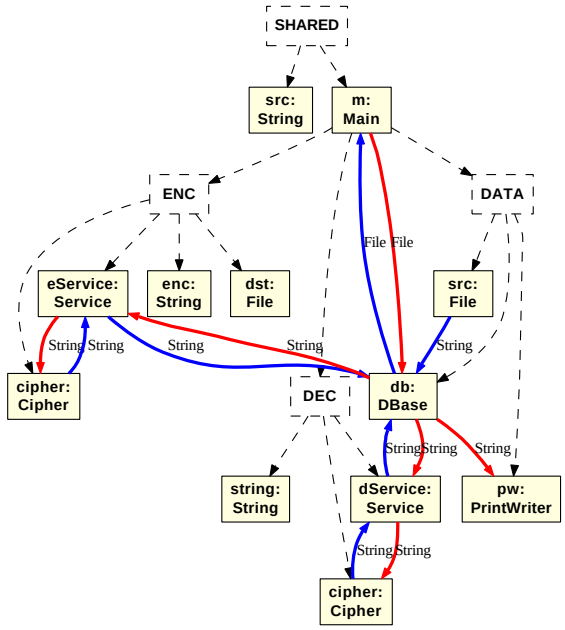
An early version of this work proposes dataflow edges that refer to types [131]. In CryptoApp, only based on the type it is unclear if the dataflow communication from `db:DBase` to `pw:PrintWriter` refers to encrypted or decrypted information for (Fig. 2.8(b)). This thesis proposes a solution where the architects can establish abstract object identity based on conceptual groups (dashed rectangles in Fig. 2.8(c)) in addition to types.

2.3 The Extracted Object Graph is Sound

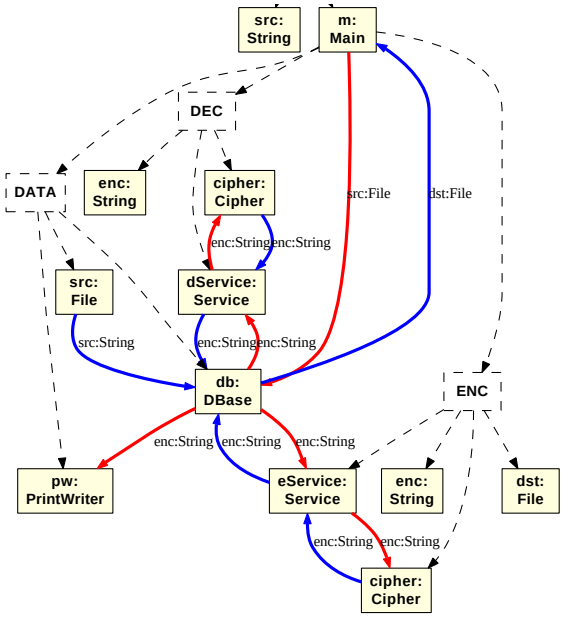
Security requires reasoning about the worst-case because one security vulnerability can compromise the whole system. Thus, an analysis needs to extract a sound approximation of the runtime architecture. For object-oriented code, an abstract object graph needs to represent all objects and relations that may exist at runtime because any object or relation can introduce security vulnerabilities. An edge on a sound object graph means "a dataflow *may* exist at runtime" rather than "a dataflow *must* exist at runtime". In fact, the dataflow might not even correspond to an actual runtime communication, due to possible false positives in the object graph when the static analysis considers all possible paths in the program, including the infeasible ones. On a sound object graph, the absence of an edge is also important because the absence means that there must be no direct communication between the two objects at runtime in any execution.



(a) A fragment of the hierarchical object graph with points-to edges [5] shows communication from `eService:Service` to `cipher:Cipher`, but miss the communication from `db:DBase` to `pw:PrintWriter`



(b) A fragment of the hierarchical object graph with dataflow edges that refer to types [131]. Since several edges refer to the type `String`, it is unclear if the object of type `String` that flows from `db:DBase` to `pw:PrintWriter` represents encrypted or decrypted information



(c) Dataflow that refer to objects show that encrypted information flow from `db:DBase` to `pw:PrintWriter`

Figure 2.8: Object graphs with points-to edges and dataflow edges that refer to types are not enough. Architects need dataflow edges that refer to objects.

Soundness

The extracted abstract object graph is sound if a mapping exists between ROG in any program run and the mapping satisfies the following conditions:

- **Object soundness.** Every runtime object has as a unique representative abstract object in the abstract graph. While one abstract object can represent multiple runtime objects, the same runtime object cannot map to two separate abstract objects. It would be misleading to have one runtime entity appear as two components in a runtime architecture. For example, one could assign the two components different values for the *TrustLevel* property and obtain invalid analysis results and a false sense of security.
- **Edge soundness.** Every runtime edge between two runtime objects has a corresponding abstract edge between the representatives of the two objects. For a runtime dataflow edge that refers to a runtime object, the corresponding abstract dataflow edge refers to the representative of the runtime object (Fig. 2.6).

An approximation of a heap snapshot is unsuitable to reason about worst-case. A dynamic analysis could conceivably extract a DFD using a snapshot of the heap. However, such a snapshot is likely to have too many details for reasoning about security. In a naive approach, an analysis could delete nodes and edges, but the resulting diagram may miss some important details, and is unsound. In contrast, a sound approach could soundly summarize the heap, ensuring that every element on the heap has a representative in the summarized diagram [83]. Such a diagram would still approximate one execution, when in fact, security analysis is a worst-case analysis that needs to consider all possible executions.

2.4 ScoriaX Considers Aliasing

Soundness also means that no one runtime object has distinct representatives in the object graph. Therefore, another challenge for the extraction analysis is to consider aliasing and determine all the objects that a variable in the code may alias. For a variable is an argument

of a method invocation, considering aliasing means identifying the abstract objects that the dataflow edge refers to. If the analysis finds different abstract objects for the same argument, it creates multiple dataflow edges from the same source to the same destination (Fig. 2.6). To preserve edge soundness, three abstract objects identify a dataflow communication edge: the source, the destination, and the abstract object the edge refers to.

One solution for supporting aliasing is to use a separate may-alias analysis provided by a static analysis framework such as SOOT [72] and WALA [65]. A may-alias analysis is fully automated, but does not scale for a large number of variables. Another solution is to use type systems such as Ownership Domains [10]. A type system is a semi-automated solution and requires type annotations. This solution is scalable because annotations are local, modular and can be checked by analyzing one class at a time. This section describes each solution in more details and discusses why ScoriaX uses a type system.

2.4.1 Using Type Systems

Many ownership type systems have been proposed for reasoning about object structures and aliasing [29, 34, 38, 94]. One such type system, Ownership Domains [10], further separates aliasing policy from its mechanism. Since there are multiple ways to express design intent, there are multiple ways to annotate a system. Ownership Domains provides the flexibility needed for architectural extraction [5].

Ownership Domains. An ownership domain is a conceptual group of objects with an explicit name and explicit policies that govern how it can reference objects in other domains. Each object is in one ownership domain that does not change at runtime. This thesis deemphasizes the explicit policies (domain links), and focuses on the aliasing mechanism. Two variables declared in different domains cannot alias. For CryptoApp, the developer declares two domains in the class `Main`, `DATA` and `LOGIC`, and places objects in these domains. A typechecker ensures that no variable declared in `DATA` is assigned to a variable declared in `LOGIC`. Otherwise, the typechecker returns a warning for the architect to address.

Domain Parameters. Domain parameters propagate ownership information of objects outside the current object. A type declaration has a class name and a list of formal domain parameters. By convention, the first element of the list is the `owner` domain. To bind formal domain parameters to actual domains, the code that instantiates the class supplies the actual domains. In this way, domain parameters allow objects to share state. Typically, a factory object does not own the objects it creates. Instead, the factory object references them through domain parameters. This way, `eAct:EActivity` can create the object `service:Service` in the domain `DATA` and the object is shared between `eAct:EActivity` and `act:ViewActivity`.

The domain `shared`. The type system also supports the domain `shared`, which has objects that can be aliased globally. Little aliasing reasoning can be done about variables declared `shared`. For example, the variable `m` of type `Main` is declared `shared`.

Ownership Domains in Java. Featherweight Domain Java (FDJ) is an extension of the core Java language that excludes advanced features such as generics, static code, exceptions, and includes domain declarations and ownership types. In FDJ, an ownership type[2] has a name of a class C and a list of domains $\overline{p}$, where p is a domain parameter α, a declared domain $n.d$ or `shared` (Fig. 2.9).

2.4.2 Using a Separate May-Alias Analysis

Existing static analysis frameworks, such as Soot [72] or WALA [65], provide may-alias analyses that determine all the objects that a variable may refer to. Sensitivity determines the precision of such analyses, which can be, for example, flow-sensitive or context-sensitive. A flow-sensitive analysis considers the order in which methods are called. In a context-sensitive analysis, the context is either a method call-site (call-site context-sensitive) or an abstract object (object-sensitive) that the receiver of a method may alias. An analysis is call-site context-sensitive if it analyzes a method invocation $r.m(a)$ multiple times based on

[2]This thesis distinguishes between a class C of a variable and the type $C<\overline{p}>$ of a variable.

$$
\begin{array}{rcll}
\mathit{cdef} & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ [\texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}] & \\
 & & \{\ \overline{\mathit{dom}};\ \ \overline{T}\ \overline{f};\ \ C(\overline{T'}\ \overline{f'},\overline{T}\ \overline{f}) & \\
 & & \{\texttt{super}(f');\texttt{this}.\overline{f} = \overline{f};\}\ \ \overline{\mathit{md}}\ \} & \textit{class declaration} \\
\mathit{dom} & ::= & [\texttt{public}]\ \texttt{domain}\ d; & \textit{domain declaration} \\
\mathit{md} & ::= & T_R\ m(\overline{T}\ \overline{x})\ \ T_{this}\ \{\texttt{return}\ e_R;\} & \textit{method declaration} \\
e & ::= & x\ \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e})\ \mid e.f\ \mid e.f = e' & \\
 & & \mid e.m(\overline{e})\ \mid \ell\ \mid \ell \triangleright e & \textit{expressions} \\
n & ::= & x\ \mid\ v & \textit{values or variable names} \\
p & ::= & \alpha\ \mid\ n.d\ \mid\ \texttt{SHARED} & \textit{domain name} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} & \textit{type: class and list of domains} \\
v,\ell,\theta & \in & \textit{locations} & \\
S & ::= & \ell\ \rightarrow\ C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) & \textit{location store} \\
\Sigma & ::= & \ell\ \rightarrow\ T & \textit{store typing} \\
\Gamma & ::= & x\ \rightarrow\ T & \textit{type environment}
\end{array}
$$

Figure 2.9: Simplified FDJ abstract syntax [10].

the call-stack of method invocations that led to $r.m(a)$. In contrast, an analysis is object-sensitive if it uses object allocation expressions to distinguish between different objects that the receiver r may alias [117]. In general, the precision of a call-site context-sensitive analysis is not comparable to that of an object-sensitive analysis [114].

Object-sensitivity. In terms of aliasing precision, the state-of-the-art analysis is object-sensitive [123, 87]. Such an analysis works well for on-demand based approaches that refine the references analyzed [118], but does not scale for a large number of references. Since in the presence of recursion, the size of the call-stack is unbounded, the analysis is parameterized with a constant k, which determines the maximum size of a sequence of object allocation expressions considered to identify an abstract object. In practice, analysis frameworks implement object-sensitive analyses for $k = 1$ or $k = 2$. For example, the Soot framework implements the object-sensitive analysis of Milanova et al. [87] for $k = 1$. That is, the implementation considers one abstract object corresponding to each object allocation expression. Smaragdakis et al. implement the *2full-1heap* object-sensitive analysis [114] with $k = 2$ by also keeping track of the call-site context. Although the precision increases, the *2full-1heap* analysis does not scale. As an alternative, they proposed an analysis that abstract objects based on types, rather than object allocation expressions, and keeps track of the type of the

```
1  class Main<owner> {
2   domain DATA,LOGIC;
3   EActivity<LOGIC,LOGIC,DATA> eAct = new EActivity();
4  }
5  class EActivity<owner,L,D> extends Activity<owner,L,D> {
6    Service<D> service = new Service();
7    void onCreate(){
8      FileManager<D> fm = (FileManager)getManager("FILE");
9      LocationManager<L> lm = (LocationManager)getManager("LOCATION");
10     Location<lent> loc = lm.getLastLocation();
11     File<lent> tempFile = fm.read("history");
12     if (tempFile.find(loc)) {
13       File<unique> encrypted = service.encrypt(tempFile);
14       Intent<unique> i = new Intent(ACTION_SEND);
15       i.putExtra(EXTRA_STREAM,encrypted);
16       Activity<L,L,D> act = new ViewActivity();
17       act.startActivity(i);
18     }
19   }
20 }
21 class ViewActivity<owner,L,D> extends Activity<owner,V,D> { ... }
22 class Activity<owner,L,D> {
23   domain MSG;
24   Intent<MSG> mIntent;
25   //cannot use lent here because the parameter is stored in a field
26   void startActivity(Intent<MSG> intent) {
27     mIntent = intent;
28   }
29   Manager<unique> getManager(String<shared> name) {
30     ...//return FileManager or LocationManager
31   }
32 }
33 class Intent<owner>{
34   void putExtra(String<shared> name, Object<owner> value) { ... }
35 }
```

Figure 2.10: CryptoApp: Code fragments with ownership types

calling context. The result is a *type-sensitive* analysis that scales and has a precision similar to *2full-1heap*.

2.4.3 ScoriaX is Domain-Sensitive

Seeking a trade-off between soundness and precision, the ScoriaX considers domains (groups of objects) as contexts and distinguishes between objects of the same type that are in different domains. Therefore, ScoriaX is *domain-sensitive* and object- and flow-insensitive. The same way that an object-sensitive analysis looks for the receiver, a domain-sensitive

analysis uses the domain of the receiver to distinguish between method invocations. For example, a domain-sensitive analysis distinguishes between the receivers `c1` and `c2` if the domain `DOM1` of `c1` is different from the domain `DOM2` of `c2` (Fig. 2.11 lines 9 and 10). However, a domain-sensitive analysis does not consider the object allocation expressions to make this distinction, and is independent of the parameter k to handle recursion. One consequence of using a predefined k is that the extracted object graphs has a predefined depth (i.e., the graph is flat for $k = 1$). Instead, a domain-sensitive analysis extracts a hierarchical object graph, where the depth of the graph is not predefined. The object hierarchy may contain cycles for recursive type declarations. Since a domain-sensitive analysis uses the aliasing precision provided by the ownership types rather than a stand-alone may-alias analysis, it avoids the scalability problems in a style similar to a type-sensitive analysis.

Domain- vs. object-sensitivity. A may-alias analysis typically merges all the objects created at the same object creation expression into one equivalence class, attaching an object label h at *each* object creation expression `new` $C()$ (line 2.1). The domain-sensitive analysis labels an object creation expression with L (line 2.2). Compared to the label h of a standard may-alias analysis, the label L is different as follows. First, multiple object creation expressions of the type C can still be represented by the same L label if the analysis maps the domain parameters to the same actual domains, whereas a basic may-alias analysis creates

```
1  class FileMngr<owner> {
2    File<owner> makeFile(String<shared> s) {
3      return new File(s);
4    }
5  }
6  class Main<owner> {
7    String<shared> s1 = "a.txt";
8    String<shared> s2 = "b.txt";
9    FileMngr<DOM1> c1 = new FileMngr();
10   FileMngr<DOM2> c2 = new FileMngr();
11   File<DOM1> f1 = c1.makeFile(s1);
12   File<DOM1> f2 = c1.makeFile(s2);
13   File<DOM2> f3 = c2.makeFile(s1);
14 }
```

Figure 2.11: Differences between object- type- and domain-sensitivity

multiple h labels. Second, if the analysis context maps the domain parameters to n different domains, then one object creation expression of type C may create n different L labels.

For example, for the same object allocation expression `new File()` in class `FileMngr` (Fig. 2.11 line 3), the domain-sensitive analysis maps the `owner` domain parameter to the `DOM1` domain in the context of `c1`, and then to `DOM2` in the context of `c2`. Therefore, the domain-sensitive analysis creates two abstract objects of type `File` for the same object allocation expression whereas a basic object-sensitive analysis ($k = 1$) creates one abstract object.

$$\texttt{new}^{h}\ C() \tag{2.1}$$

$$\texttt{new}^{L}\ C{<}p_{owner}, p_{params}...{>}() \tag{2.2}$$

Domain- vs. type-sensitivity. A type-sensitive analysis for $k = 2$ identifies the objects based on three types: the type of the allocated object, the type of allocator object, and the type of the receiver that instantiates the allocator object. For example, the type-sensitive analysis would distinguish between different objects of type `File` only if `FileMngr` were to be instantiated in different classes `Main1` and `Main2`. For one class `Main` with at least two domains, e.g., `DOM1` and `DOM2`, the domain-sensitive analysis is more precise than the type-sensitive analysis.

2.5 ScoriaX Supports Legacy Code

Legacy code can have a high business value, and vulnerabilities in legacy code are expensive [57]. ARA needs to support legacy code instead of requiring programs to be re-engineered to a radical language that guarantees some property by construction [11]. Re-engineering is usually non-trivial, and research languages do not enjoy the same level of tool support and libraries as mainstream programming languages. Another option is for the program to

be implemented with specific libraries, or to generate code from a detailed model such as SecureUML [80]. The later might be problematic because as the system evolves developers are required to maintain consistency between the model and the code.

One limitation of the current ARA process is that developers manually draw DFDs. Since it is difficult for a developer to remember all the details, a manually drawn DFD may miss important components or connectors that may exist at runtime, and a security vulnerability related to a missed connector may not be exposed. Instead, ARA needs to use a sound approximation of a runtime architecture and one way to achieve soundness is to use a static analysis that automatically extracts a sound approximation from the code. ScoriaX supports legacy code, requiring only annotations. Fig. 2.12 shows the code of CryptoApp using Ownership Domain annotations implemented using Java 1.5 annotations where the annotations values are provided as string constants and are typechecked to ensure that annotations are consistent with each other and with the code [4, Appendix A]. The rest of the thesis uses the less verbose FDJ syntax (Fig. 2.9).

2.5.1 The Extracted Object Graph has Traceability to Code

On a manually drawn architecture, the architects spend additional effort reading the code and investigating each potential architectural flaw they find. This effort can be reduced if the architectural model has traceability to code. Finding a vulnerability using the extracted object graph allows the architects to trace directly to the corresponding lines of code that may introduce the vulnerability. The extracted object graph is consistent with code and every node and edge has traceability information. Architects can trace from each abstract object to one or more object allocation expressions, and from each dataflow edge to one or more field read, field write or method invocation expressions in the code.

Since the architects may need more than one expression to understand a reported vulnerability, ScoriaX also keeps track of traceability paths. A traceability path is a sequence of expressions that may lead to the creation of an object or a dataflow edge. The traceability

```
@DomainParams({"owner"})
@Domains({"DATA","LOGIC"}}
class Main {
 @Domain("LOGIC<LOGIC,DATA>") EActivity eAct = new EActivity();
}

@DomainParams({"owner","L","D"})
@DomainInherits({"Activity<owner,L,D>"})
class EActivity extends Activity {
  @Domain("D") Service service = new Service();
  void onCreate(){
    @Domain("D") FileManager fm = (FileManager)getManager("FILE");
    @Domain("L") LocationManager lm = (LocationManager)getManager("LOCATION");
    @Domain("lent") Location loc = lm.getLastLocation();
    @Domain("lent") File tempFile = fm.read("history");
    if (tempFile.find(loc)) {
      @Domain("unique") File<unique> encrypted = service.encrypt(tempFile);
      @Domain("unique") Intent i = new Intent(ACTION_SEND);
      i.putExtra(EXTRA_STREAM,encrypted);
      @Domain("L<L,D>") Activity act = new ViewActivity();
      act.startActivity(i);
   }
  }
}
@DomainParams({"owner","L","D"})
@DomainInherits({"Activity<owner,L,D>"})
class ViewActivity extends Activity { ... }

@Domains("MSG")
@DomainParams({"owner","L","D"})
class Activity {
  @Domain("MSG") Intent mIntent;
  //cannot use lent here because the parameter is stored in a field
  void startActivity(@Domain("MSG") Intent intent) {
    mIntent = intent;
  }
  @Domain("unique") Manager getManager(@Domain("shared") String name) {
    ...//return FileManager or LocationManager
  }
}

@DomainParams({"owner"})
class Intent {
  void putExtra(@Domain("shared") String name, @Domain("owner") Object value) { ... }
}
```

Figure 2.12: CryptoApp: Code fragments with Ownership Domain annotations.

path allows architects to understand why the analysis creates an abstract object and why a vulnerability exists. Since ScoriaX does not use a precomputed call graph, the sequence consists of several object allocation expressions for abstract objects. For dataflow edges, a

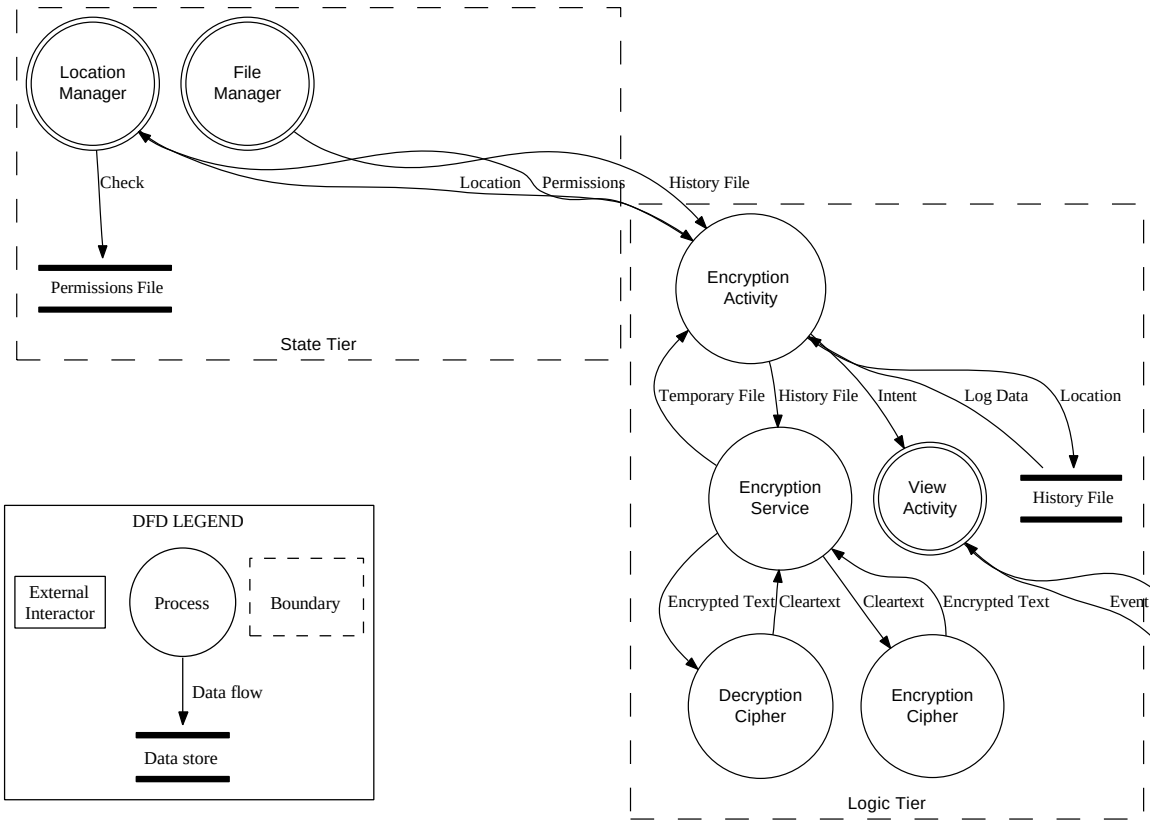


Figure 2.13: Data-Flow Diagram supports hierarchical decomposition and shows details of the high-level process *Encryption Service*.

traceability path starts with a sequence of object allocation expressions and ends with a method invocation, a field read or a field write expression.

2.6 The Extracted Object Graph is Hierarchical

ARA requires a diagram that fits on one page [86, Chapter 5]. To satisfy this requirement, a runtime architecture uses hierarchical decomposition. Hierarchy enables high-level understanding and detail such that in a Level-i DFD a *HighLevelProcess* can be expanded into a Level-(i+1) DFD to show more details. For example, by expanding the component *Service*, the CryptoApp DFD-L2 (Fig 2.13) shows two distinct components of the same type: the *Encryption Cipher* that encrypts data, and the *Decryption Cipher* that decrypts data.

2.6.1 Flat Object Graph

Without a hierarchical organization of objects, a flat object graph mixes low-level objects that are data structures such as `HashMap` with architecturally relevant objects from the application domain such as `eAct:EActivity`. The architects have no easy way to distinguish between these objects. One reason that most analyses extract flat object graphs is that architectural hierarchy is not readily observable in code that is written in a general purpose programming language. Existing approaches that extract flat object graphs are either unsound [67, 116], sound but with only points-to edges [96], or show no labels on the dataflow edges [87].

Flat object graph constraints. If architects were to reason about security based on a sound, flat object graph with dataflow edges that refer to objects, the best architects can express is objects that transitively flow from a source to a destination and object reachability (Section 5.2, page 108). Object hierarchy increases the precision and allows architects for example to distinguish objects of the same type based on the substructure of an object. For example, `SocketOutputStream` that is in the substructure of `socket:Socket` has the same type as another object in the substructure of `sslSocket:SSLSocket`. If the communication is provided by an object of type `Socket` that does not provide encryption, the confidential data is vulnerable to eavesdropping (Fig. 2.14). A constraint that checks direct communication to the object of type `Socket` does not find this vulnerability because the password is sent indirectly to a descendant. If the architects were to assume that all objects of type `SocketOutputStream` are untrusted, the approach would return a false positive for the object of type `SocketOutputStream` that is a descendant of an object of type `SSLSocket`, which does provide encryption.

```
1  class Host{
2    Socket socket = new Socket("localhost", 9999);
3    SSLSocket sslSocket = ...;
4    void sendPassword1(String pwd) {
5      OutputStream sout = socket.getOutputStream();
6      PrintWriter out = new PrintWriter(sout, true);
7      out.println(pwd);
8    }
9    void sendPassword2(String pwd) {
10     OutputStream sout = sslSocket.getOutputStream();
11     PrintWriter out = new PrintWriter(sout, true);
12     out.println(pwd);
13   }
14 }
```

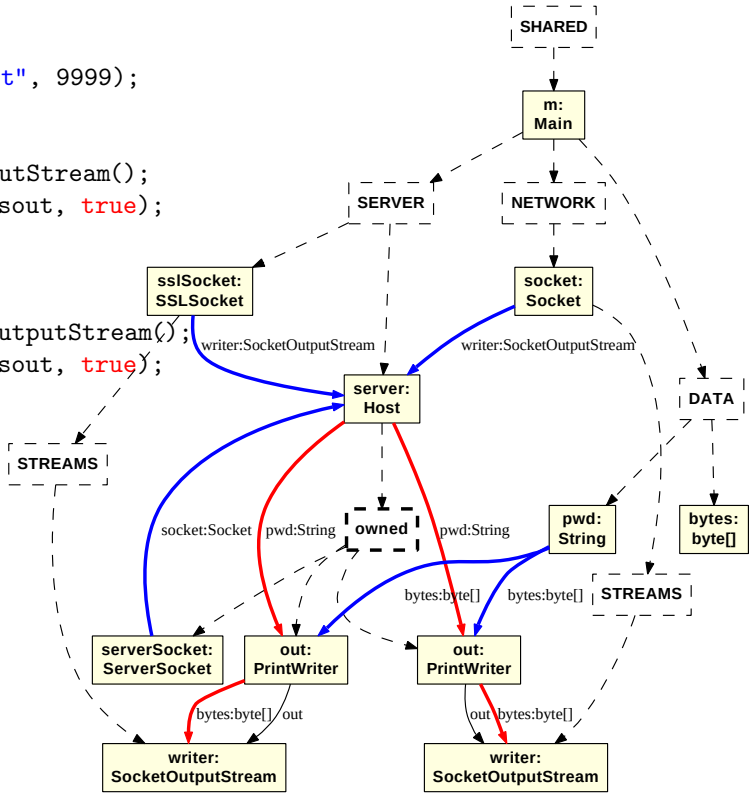


Figure 2.14: By using indirect communication, Scoria reports that sending the password to a descendant of an object of type `Socket` makes the password vulnerable to eavesdropping. The password should be sent using an object of type `SSLSocket`.

2.6.2 Object Hierarchy with Ownership Domains

In the abstract object hierarchy, an abstract object can have domains, and in turn, each domain can have abstract objects. A domain is declared on a class, but is treated like a field in the sense that a fresh domain is created for each instance of that class. For a domain `MSG` declared on a class `Activity`, and two instances `o1` and `o2` of type `Activity`, the domains `o1.MSG` and `o2.MSG` are distinct for distinct `o1` and `o2`. An object does not have child objects directly. Instead, an object has domains, which in turn have objects.

The data type OGraph describes the representation of an abstract object graph that ScoriaX extracts (Fig. 2.15). The OGraph has nodes that represent abstract objects (OObjects) and group of objects (ODomains), and edges that represent dataflow communication between abstract objects (OEdges). The OGraph has a set of abstract objects DO. To represent the hierarchy, the OGraph uses a mapping DD from a domain declaration in the context of an

$$
\begin{array}{ll}
G \in \mathsf{OGraph} & ::= \langle\ \textbf{Objects} = DO,\ \textbf{DomainMap} = DD,\ \textbf{Edges} = DE\ \rangle \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Type} = C{<}\overline{D}{>}\ \rangle \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst},\ \textbf{Label} = O_{label}, \textbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle \\
DD & ::= \emptyset\ \mid\ DD \cup \{\ (O, C{::}d) \mapsto D\ \} \\
DO & ::= \emptyset\ \mid\ DO \cup \{\ O\ \} \\
DE & ::= \emptyset\ \mid\ DE \cup \{\ E\ \} \\
\Upsilon & ::= \emptyset\ \mid\ \Upsilon \cup \{\ C{<}\overline{D}{>}\ \}
\end{array}
$$

Figure 2.15: Data type declarations for the abstract object graph (OGraph).

OObject $(O, C{::}d)$ to an abstract domain D. To keep track of the OObjects created from the root of the OGraph, the analysis uses a stack Υ. This stack ensures that the analysis terminates in the presence of recursive types (Section 2.8.1). The OGraph is a multi-graph, where multiple edges with different labels might exists between the same source and destination. The OGraph also has a set of dataflow edges DE, where each OEdge is identified using a source, a destination, the OObject that the dataflow refers to, and a directionality flag. The import flag indicates that an object reads data from a known source, while the export flag indicates that an object writes data to a known destination.

Several key features of Ownership Domains are crucial for expressing design intent in code. The first is having explicit "contexts" or domains. Other ownership type systems implicitly treat all objects with the same owner as belonging to one implicit context [111]. For architectural extraction, explicit domains are useful, because developers can define multiple domains per object to express their design intent. For CryptoApp, the architects define two top-level domains or tiers, `DATA` and `LOGIC` and places objects in these domains (Fig. 2.10).

Second, Ownership Domains support pushing any object underneath any other object in the ownership hierarchy. A child object may or may not be encapsulated by its parent object. Ownership Domains defines two kinds of object hierarchy: logical containment using public domains and strict encapsulation using private domains.

For `HashMap`, its representation is an array `table` of elements of type `Entry`. The array

```
1  //K, V generic types
2  class Entry<OWNER K<DOMKEY>, V<DOMVALUE>> {
3    K<DOMKEY> key; // Virtual field
4    V<DOMVALUE> value; // Virtual field
5  }
6  class HashMap<OWNER, K<DOMKEY>, V<DOMVALUE>> {
7    domain OWNED;
8    public domain ENTRIES,VALSET,KEYSET;
9    Entry<ENTRIES,K<DOMKEY>,V<DOMVALUE>> entry;
10   Entry<OWNED[ENTRIES,DOMKEY,DOMVALUE]>[] table;
11   Collection<VALSET V<DOMVALUE>> values() { ... }
12   Set<KEYSET K<DOMKEY>> keySet() { ... }
13   Set<ENTRIES Entry<ENTRIES K<DOMKEY>, V<DOMVALUE>>> entrySet() { ... }
14 }
```

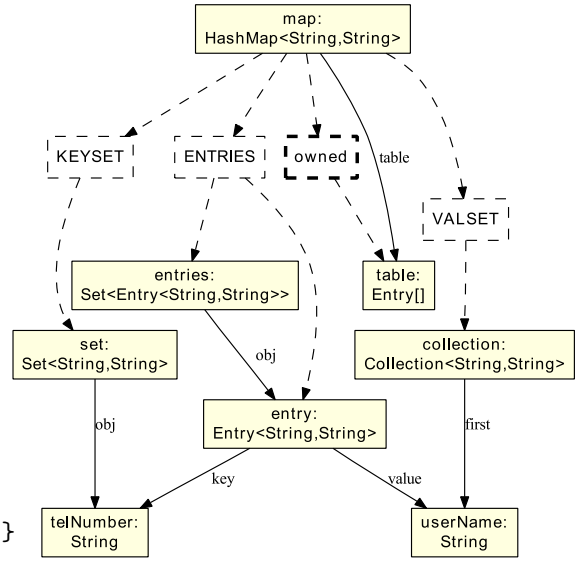
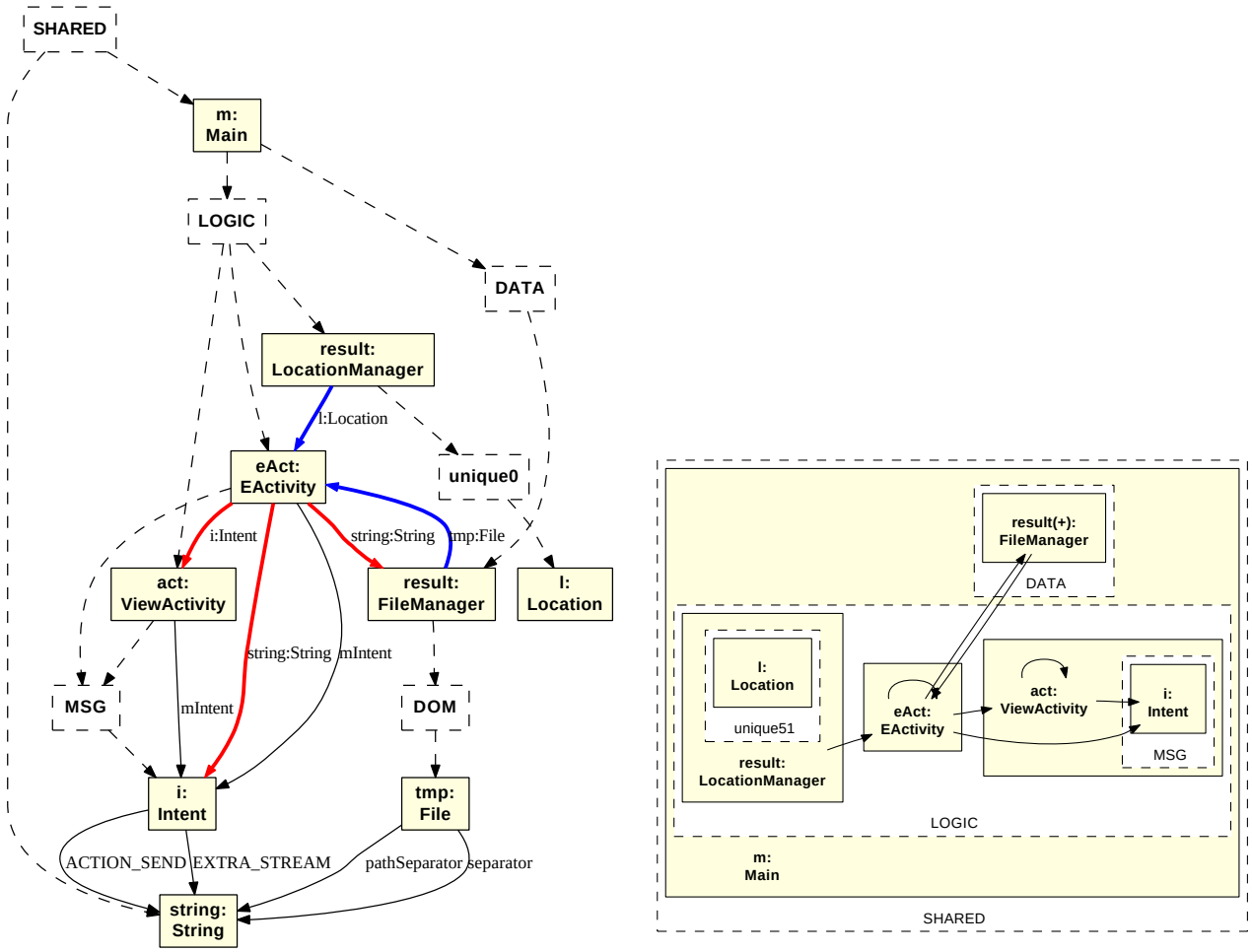


Figure 2.16: An object of type `HashMap` has a private domain that protects its representation and several public domains to store the entries, keys and values accessible by clients.

is in the private domain `OWNED` because the object `table:Entry[]` should never be directly referenced from outside its owner because if the map grows, the array is replaced with a different larger array. On the other hand, the method `entrySet` returns a set of objects of type `Entry` that is in the public domain `ENTRIES`. The client code needs access to this set if it were to copy the content of the map (Fig. 2.16).

OGraph graphical notation. Figure 2.17 shows a fragment of an OGraph such that architecturally relevant objects are shown near the top of the figure and implementation details further down. A filled rectangle represents an object, a thin dashed rectangle represents a public domain, and a thick dashed rectangle represents a private domain. For edges, a thin dashed edge represents a parent-child relation, a thick edge represents a dataflow, and a thin edge represents a points-to relation.

OGraph Visualization A hierarchical representation allows expanding or collapsing the substructure of selected objects to control the level of visual detail. For example, only



(a) Fragment of the CryptoApp OGraph. (b) Collapsed substructure.

Figure 2.17: The CryptoApp OGraph shows a hierarchy of objects and domains, and dataflow edges between objects. Using nested boxes can make the visualization less cluttered.

the substructure of `act:ViewActivity` and `result:LocationManager` is visible, while the substructures of `eAct:EActivity` and `result:FileManager` are collapsed (Fig. 2.17(b)). A (+) symbol indicates than an object has a collapsed substructure. While collapsing, the visualization also lifts any edge that might be incoming to or outgoing from a child object, i.e., the edge is recursively lifted to the nearest visible ancestor of the child object. The nested box visualization can show only the objects at the top level, which makes the graph less cluttered. This thesis focuses on the underlying representation of the abstract graph and constraints. Lifting edges and collapsing objects are related to visualization [120] and are addressed elsewhere [4, Sec. 2.4.3].

2.7 ScoriaX Supports Extensions of Ownership Domains

The result of the analysis is a sound, hierarchical object graph with dataflow edges (in addition to the previously extracted points-to edges) which can be used to find security vulnerability such as information disclosure. In the OGraph, each abstract object is in a named, conceptual group. Extensions of Ownership Domains allow an object to be borrowed from one domain to another, and the declaration of the variable that aliases this object is annotated `lent`. If only one variable can alias an object at a time (an object is passed linearly), then these variables are annotated `unique`. Previous work [5, 4] extracted only points-to edges and did not handle `lent` and `unique`. To extract dataflow edges, the analysis needs to handle expressions that involve local variables or method parameters. ScoriaX resolves `lent` and `unique` using a domain-sensitive value flow analysis and attempts to place borrowed objects and objects passed linearly to an ODomain. If it fails, it creates special domains and creates *flow objects*.

If the analysis were to ignore the extensions of Ownership Domains, the analysis would fail to show representatives for some of the runtime objects and edges and the extracted graph would be unsound. Consider for example the method `getManager` (Fig. 2.12, line 29) that returns a value of type `Manager<unique>`. When the analysis traverses the code of class `Activity`, which declares the method, the domain is unknown and depends on the context in which the method is invoked. The method `getManager` is invoked twice in the class `EActivity` where the return value is assigned to a variable of type `FileManager<D>`. Hence, the analysis needs to resolve `unique` to `D` and create the object `result:FileManager` in `DATA`. For the second invocation, the return value is assigned to `LocationManager<L>` and the analysis needs to resolves `unique` to `L` and create the object `result:LocationManager` in `LOGIC`. I will now discuss how the analysis works in more detail.

$$
\begin{array}{rcll}
CT & ::= & \overline{cdef} & \textit{table of class declarations} \\
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} & \\
 & & \{\ \overline{dom};\ \ \overline{T_A}\ \overline{f};\ \ C(\overline{T'_A}\ \overline{f'},\overline{T_A}\ \overline{f}) & \\
 & & \{\texttt{super}(f');\texttt{this}.\overline{f}=\overline{f};\}\ \ \overline{md}\ \} & \textit{class decl.} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; & \textit{domain decl.} \\
md & ::= & T_A\ ret\ m(\overline{T_B}\ \overline{x})\ \ T_{this}\ \{ret = e_R;\texttt{return}\ ret;\} & \\
 & & & \textit{method decl.} \\
e & ::= & \mid x = \texttt{new}\ C\texttt{<}A,\overline{p}\texttt{>}(\overline{y}) & \\
 & & \mid x = y.f\ \mid x.f = y \mid x = y\ \mid x = r.m(\overline{y}) & \\
 & & \mid \ell\ \mid \ell \triangleright e & \textit{expressions} \\
n & ::= & x \mid v & \textit{values or variable names} \\
p & ::= & \alpha \mid n.d \mid \texttt{SHARED} & \textit{domain name} \\
A & ::= & \texttt{unique} \mid p & \textit{domain may be unique} \\
B & ::= & \texttt{lent} \mid A & \textit{domain may be lent or unique} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} & \textit{precise type} \\
T_A & ::= & C\texttt{<}A,\overline{p}\texttt{>} & \textit{owner domain may be unique} \\
T_B & ::= & C\texttt{<}B,\overline{p}\texttt{>} & \textit{owner domain may be lent or unique} \\
v,\ell,\theta & \in & \textit{locations} & \\
x,y,r,a & \in & \textit{variables} & \\
S & ::= & \ell \rightarrow C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) & \textit{location store} \\
\Sigma & ::= & \ell \rightarrow T & \textit{store typing} \\
\Gamma & ::= & x \rightarrow T_B & \textit{type environment}
\end{array}
$$

Figure 2.18: Three-address code version of the FDJ syntax extended using `lent` and `unique` [12]

2.7.1 Extensions of Ownership Domains

This section provides a revised abstract syntax of FDJ (Fig. 2.18) that supports the Ownership Domain extensions, gives an informal definition of flow objects, discusses precision tradeoffs, and describes the algorithm ScoriaX uses to extract the `OGraph`.

The `unique` annotation. A variable declared `unique` refers to an object to which there is only one reference, and can be passed linearly from one domain to another. For example in the factory method `getManager`, only the client code knows the actual domain of the object created by the factory (Fig. 2.10, line 29). Therefore, the method returns `unique` objects of various types that extend `Manager`. Next, the client code in `EActivity` assigns these objects to the domain parameter `D` for `FileManager` object and `L` for `LocationManager` object (Fig. 2.10, line 8, 9). The analysis uses the type of the sink variable to resolve unique to `D` or `L`.

The `lent` annotation. A variable declared `lent` refers to an object that is temporarily borrowed from one domain to another as long as an object in the second domain does not create a persistent reference to the borrowed object, e.g., by storing it in a field. Only method formal parameters and local variables can be `lent`. For example, local variable `tempFile` (line 11), and the formal parameter `x` (line 13) are declared `lent`.

2.7.2 Other Flow Objects

The core of the analysis uses the ownership types to find objects that a dataflow refers to. In the extension of Ownership Domains, a method invocation may involve variables declared `lent` and `unique`. Therefore, some of the objects are in a domain that does not correspond to a domain declaration, while other objects may be borrowed between domains. The extraction analysis tracks the value flow and attempts to find an actual domain from where the object is borrowed, or an actual domain where the object flows into. If it succeeds, we say that the analysis resolves `lent` and `unique` to an actual domain. Otherwise, the analysis creates a special domain as a child of the creator OObject with an automatically generated name and without a domain declaration. Then, the analysis creates a *flow object* in this special domain. An example of a flow object is the abstract object `l:Location` which flows from `result:LocationManager` to `eAct:EActivity` (Fig. 2.17).

2.7.3 Value Flow Graph

The analysis may also encounter the annotations `lent` and `unique` from extensions of Ownership Domains [12], which do not map directly to actual domains. The annotation `lent` is used when an object is borrowed from one domain to another. A variable with any type of annotation can be assigned to a variable declared `lent`, which is a universal sink. ScoriaX keeps track of assignments between variables to resolve `lent` to the domain of the source. The annotation `unique` is used when an object is passed linearly from one domain to another without having more than one persistence reference to it. A variable declared

`unique` is a universal source and can be assigned to a variable with any type of annotation. ScoriaX resolves `unique` to the domain of the destination.

To resolve `lent` and `unique`, the analysis builds a domain-sensitive value flow graph that has different nodes and edges than the abstract object graph. A value flow node is a triplet composed of a variable, the type of the variable with the owner domain and the abstract object that represents the context in which a variable is used. By considering O as a part of the value flow node, the value flow analysis is also domain-sensitive and has different nodes for the same variable x analyzed in different contexts. The analysis attempts to resolve `lent` and `unique` to a declared domain, a domain parameter or `shared`. To resolve `lent`, the analysis finds the node of the variable x in a given context O and goes one step backward in the value flow. To resolve `unique`, the analysis goes one step forward in the value flow. If the analysis fails to resolve `lent` or `unique`, it generates a special domain as a child of the creator object, and places the object in this special domain. We call such an object *a flow object*. The intuition behind such an object is that it flows only along edges and does not always reside in one domain.

A value may flow through several consecutive assignments, and the analysis uses a transitive flow to resolve `lent` and `unique`. Computing a simple transitive closure of the value flow graph provides a sound but imprecise solution. To achieve precision and keep track of call-site context sensitivity [76, 75], the value flow analysis uses labels on the value flow edges. For assignments and field reads, $\bullet$ means no label. For method invocations, the analysis generates a fresh value of i for an invocation $x = y.m(a)$ in the context of O_{this}. To be call-site context-sensitive, the analysis matches the parentheses with the same value of i to summarize the effect of a method invocation. The analysis adds value flow edges to track the assignments of arguments to formal method parameters and the assignment of the return value to the left-hand-side of the method invocation expression. For field write expressions, the analysis uses the $\star$ label that cancels the effect of a $(_i$ label because a method m_i can store a value passed as an argument into a field, and the value can be returned by

$$FG \quad ::= \emptyset \mid FG \cup \{\ (O_{src}, x, B_{src}) \overset{label}{\rightsquigarrow} (O_{dst}, y, B_{dst})\ \} \quad \textit{Value Flow Graph}$$
$$label \quad ::= (_i \mid)_i \mid \bullet \mid \star \quad \textit{value flow labels}$$

$x = y.m(a)$	$(O_{\texttt{this}}, x, B_x) \overset{(_i}{\rightsquigarrow} (O_y, \texttt{this}, \texttt{owner})$	1 `// this` $\mapsto O_{\texttt{this}}$
	$(O_{\texttt{this}}, a, B_a) \overset{(_i}{\rightsquigarrow} (O_y, fa, B_{fa})$	2 `domain DOM;`
	$(O_y, ret, B_r) \overset{)_i}{\rightsquigarrow} (O_{\texttt{this}}, x, B_x)$	3 `y = new C<DOM>();`
	$i ::= fresh_i(O_{\texttt{this}}, x = y.m(a))$	4 `X<Bx> x = y.m(a);`
$x = y.f$	$(O_y, f, B_f) \overset{\bullet}{\rightsquigarrow} (O_{\texttt{this}}, x, B_x)$	5 `class C<owner> {`
$x.f = y$	$(O_{\texttt{this}}, y, B_y) \overset{\star}{\rightsquigarrow} (O_x, f, B_f)$	6 `//this` $\mapsto O_y$
$x = y$	$(O_{\texttt{this}}, x, B_x) \overset{\bullet}{\rightsquigarrow} (O_{\texttt{this}}, y, B_y)$	7 `//`$(O_y, C$`::owner)` $\mapsto$ `DOM`
		8 `X  m(A<Bfa> fa) {...`
		9 `return ret;`
		10 `}`
		11 `}`

Figure 2.19: ScoriaX tracks assignments and extracts the value flow graph (FG)

another method m_j (Fig. 2.19). The analysis then uses a worklist algorithm and the labels to compute the transitive flow and concatenates two value flow edges where the first edge has the same destination and the source of the second edge (Section 3.3.4, page 81).

2.7.4 ScoriaX Algorithm

Since the analysis uses an OObject as a part of the value flow node, the constructions of the value flow graph and OGraph are inter-dependent. Therefore, the analysis does multiple iterations over the code starting from the root expression, and builds up upon the OGraph and flow graph created in the previous pass. At each iteration, the analysis starts from the root expression and stops at a fixed point when it can no longer add nodes or edges in the flow graph and OGraph. After initialization, the analysis first creates objects, domains and dataflow edges, and extracts the value flow graph FG. Next, the analysis computes the transitive flow FG_P only for the references declared `lent` and `unique`. In the last iteration, the analysis uses the transitive flow to extract more edges and flow objects (Fig. 3.18).

Instead of a transitive closure of FG the value flow analysis uses the most precise type

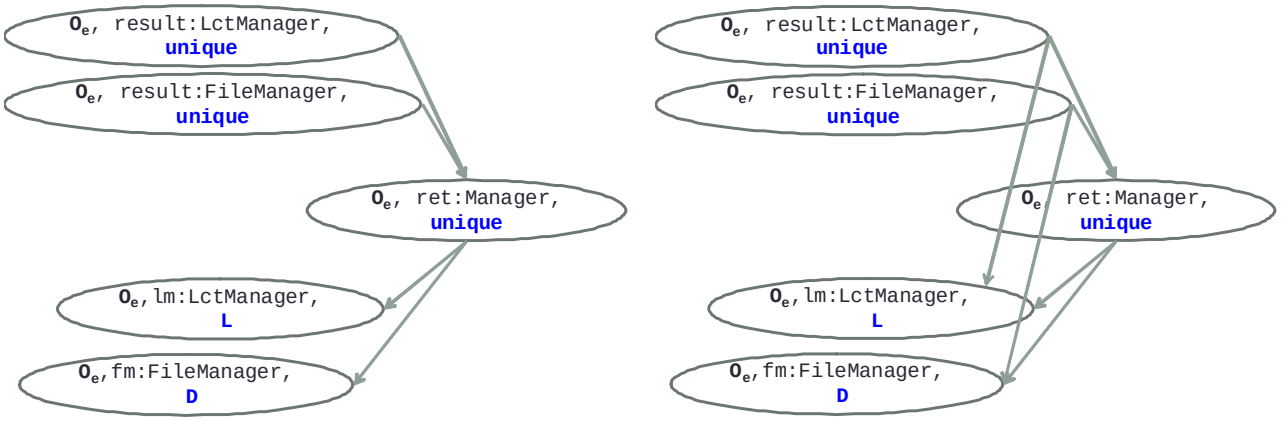
Initialization:
$DO = DO_0$, $DD = DD_0$
$DE = DE_0$ $FG = FG_0$, $\Gamma = \emptyset$, $\Upsilon = \emptyset$
$G = \langle DO, DD, DE \rangle$ $G' = \langle DO', DD', DE' \rangle$
$G' \subseteq G \Leftrightarrow DO' \subseteq DO \wedge DD' \subseteq DD \wedge DE' \subseteq DE$
Pass1: Extract OObjects (DO), ODomains, and construct DD
Pass2: Extract dataflow edges (DE) and value flow graph (FG)
Pass3: Summarize value flow graph and compute transitive flow
$FG* = summarize(FG)$
$FG_P = propagateAll(FG)$
Pass4: Attempt to resolve `lent` and `unique` using transitive flow (FG_P)
Extract more dataflow edges OEdges in DE and flow objects in DO
At every pass:
Start from e_{root} in the context of O_{world}
Stop when a fixed point is reached

Figure 2.20: The analysis iterates multiple times and stops at a fixed point

of the variables to create the transitive flow graph. In the running example (Fig. 2.12), the analysis needs to resolve `unique` for objects of type `FileManager` and `LocationManager` in the context of the object `eAct:EActivity` (O_e). The value flow graph contains edges from $(O_e, \texttt{result}, \texttt{unique})$ to $(O_e, \texttt{ret}, \texttt{unique})$ and further to $(O_e, \texttt{lm}, \texttt{L})$, where the class of `result` and `lm` is `LocationManager`, and of `ret` is `Manager` (Fig. 2.21(a)). To find the actual domain, the analysis resolves `unique` to the domain parameter `L` going forward in the transitive flow graph (Fig. 2.21(b)) and creates the object `result:LocationManager` in `LOGIC` because the domain parameter `L` maps to `LOGIC` in DD. Similarly, the analysis creates the object `result:FileManager` in `DATA`. If the analysis were to simply create the transitive closure, it would have resolved `unique` to `L` and `D` and would have consequently created two false positive objects, `result:FileManager` in `LOGIC` and `result:LocationManager` in `DATA`, which might have further led to a spurious number of false positive edges.

2.7.5 Value Flow Analysis is Domain-Sensitive

The value flow analysis is domain-sensitive and creates multiple value flow nodes for the same variable. If ScoriaX were to use a domain-insensitive value flow graph, the object graph would have had false positive edges, which would have further led to false positives



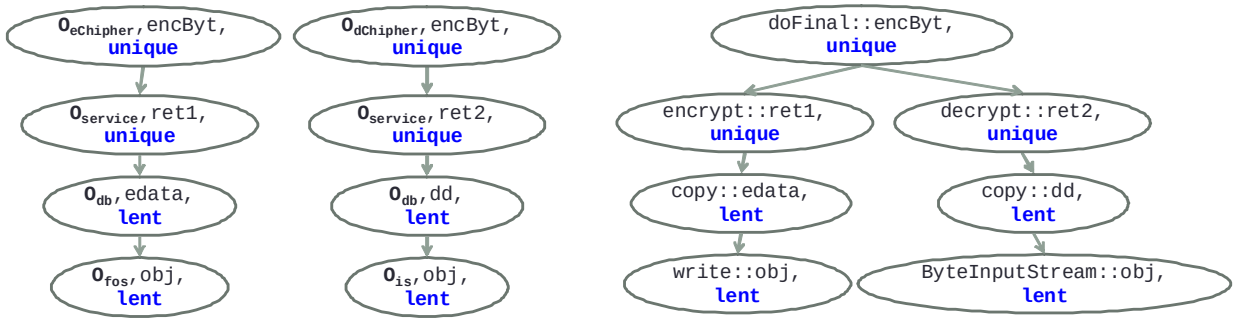
(a) Value flow graph (FG) (b) Transitive flow (FG_P)

Figure 2.21: . Fragments of value flow graph the analysis uses to resolve `unique` for variable of type `Manager`. The analysis does not simply create the transitive closure of FG, but uses the declared type of variables to create the transitive flow.

reported by constraints. The code fragments in Fig. 2.22 are from an application that uses encrypted account information from a file. The method `copy` reads the content of the file, decrypts the information, and creates an encrypted backup. The method uses a service that has both an encryption and a decryption cipher. To ensure that the password is not stored in clear text, the architect formulates the constraint: the same object that flows from the decryption cipher to the encryption service does not flow to a file in clear text. The code uses `unique` annotations for the return types of `encrypt` and `doFinal`. SocioriaX extracts the object `eCipher:Cipher` used for encryption and `dCipher:Cipher` used for decryption because they are in different domains. A domain-sensitive value flow shows that only encrypted data flows into the parameter `obj` of the method `write` in the context of `fos:FileOutputStream` (Fig. 2.23(a)). The decrypted data only flows to the parameter `obj` in the context of `is:ByteInputStream`. On the other hand, a domain-insensitive value flow graph excessively merges value flow nodes for the variable `encByt` in the method `doFinal`, and shows a false positive transitive flow as if decrypted data is written in a file (Fig. 2.23(b)).

```
1  class Service<owner> {
2   domain ENC,DEC;
3   Cipher<ENC> eCipher = new Cipher();
4   Cipher<DEC> dCipher = new Cipher();
5   String<unique> encrypt(String<lent> t) {
6    return eCipher.doFinal(t);
7   }
8   String<unique> decrypt(String<lent> t) {
9    return dCipher.doFinal(t);
10  }
11 class Cipher<owner> {
12  String<unique> doFinal(String<lent> t){
13   String<unique> encByt = new String();
14   ... //simplified model of encryption
15   return encByt;
16  }
17 }
18 class DBase<owner,L> {
19  domain SRC,DST;
20  Service<L> service = new Service();
21  void copy(File<lent> src, File<lent> dst) {
22   String<SRC> mydata = readContent(src);
23   String<lent> dd = service.decrypt(mydata);
24   InputStream<owner> is = new ByteInputStream(dd);
25   ...
26   OutputStream<DST> fos = new FileOutputStream(dst);
27   String<owner> data = ...;
28   String<lent> eData = service.encrypt(data);
29   fos.write(eData);
30  }
31 }
```

Figure 2.22: Object provenance: the object of type `Service` has two ciphers one for encryption one for decryption. Encrypted and decrypted data corresponds to the same object allocation expression in the code. Security constraint: *the same data that flows from decryption cipher to `DBase` does not flow back into the file without encryption.*



(a) A domain-sensitive value flow distinguishes between encrypted and decrypted data created by different ciphers that are in different domains.

(b) A domain-insensitive value flow graph excessively merges value flow nodes and shows that a decrypted value may flow to the method `OutputStream.write`.

Figure 2.23: Fragments of the value flow graph for the code in Fig. 2.22.

2.8 ScoriaX Summarizes Runtime Object Graphs

If the architects were to reason about security based on an object diagram or a runtime object graph, they would need to apply the same constraint on many similar instances. The challenge is to summarize an unbound representation, such as an ROG, with an abstract object graph that is finite. In the OGraph, an OObject is a canonical object that merges n possible runtime objects of the same class, where n is unbounded. By addressing the soundness challenge, the analysis ensures that every runtime object has a unique representative in the OGraph.

2.8.1 Recursive Types

The analysis must handle recursive types that would otherwise lead to an unbounded number of nodes. To get a finite OGraph and ensure that the analysis terminates, the analysis could stop expanding the OGraph after a certain depth. Truncating the recursion at an arbitrary depth may omit objects or edges beyond the cutoff depth, which would be unsound. Instead, to preserve soundness, the analysis creates a cycle in the OGraph when it encounters a domain declaration (C::d), already analyzed in the context of the same OObject O_C.

A domain declaration may be analyzed multiple times using different contexts. As a result, the OGraph has multiple ODomains for the same domain declaration. On the first pass, the analysis creates the OObject $O_C = C\langle\overline{D}\rangle$, adds it in the stack Υ, and performs a lookup in the domain map DD. If the pair $(O_C, C::d)$ is not found, it creates a new ODomain and a new entry for the pair in DD. On the second pass, the analysis finds the same pair and reuses the existing ODomain. So the analysis creates a cycle in the OGraph, and the reused ODomain appears as the child of two OObjects. This justifies an ODomain not having a unique owning OObject (Fig. 2.15). On the next pass, the analysis encounters the same $C\langle\overline{D}\rangle$ in Υ and does not recurse any further.

```
1  Main<SHARED> main = new Main<SHARED>();
2  class Main<OWNER> {
3    domain OWNED;
4    QuadTree<this.OWNED> aQT;
5    Main() {
6      this.aQT = new QuadTree<this.OWNED>();
7    }
8  }
9  class QuadTree<M> {
10   domain OWNED;
11   QuadTree<OWNED> nwQT;
12   QuadTree() {
13     this.nwQT = new QuadTree<this.OWNED>();
14   }
15 }
```

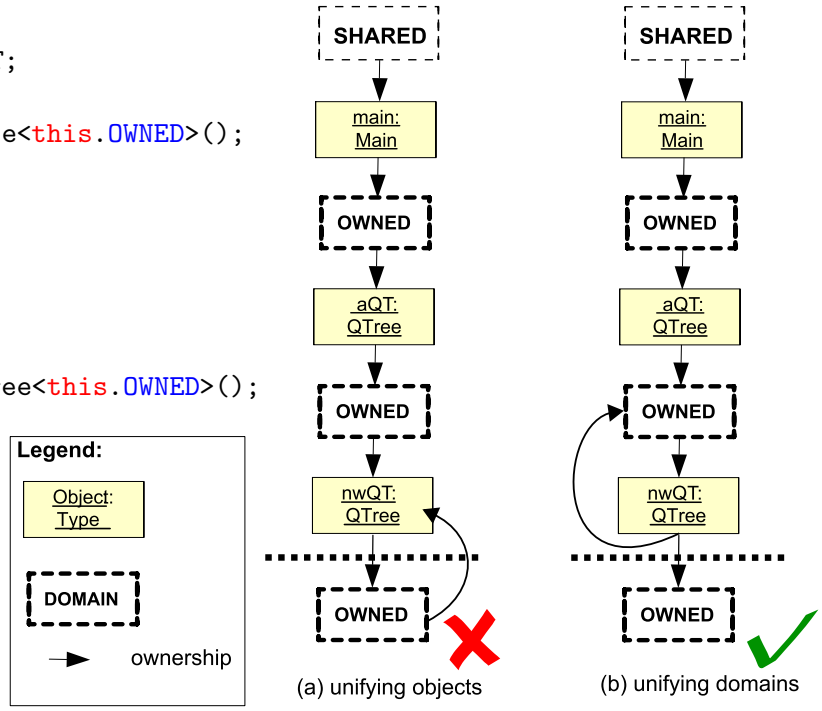


Figure 2.24: Handling recursive types, revised from [4, Figure 2.22].

```
1  Main<SHARED> main = new Main<SHARED>();
2  class Main<OWNER> {
3    domain OWNED;
4    QuadTree<this.OWNED> aQT;
5    Main() {
6      this.aQT = new QuadTree<this.OWNED>();
7    }
8  }
9  class QuadTree<M> {
10   domain LEFT, RIGHT;
11   QuadTree<LEFT> left;
12   QuadTree<RIGHT> right;
13   QuadTree() {
14     this.left = new QuadTree<this.LEFT>();
15     this.right = new QuadTree<this.RIGHT>();
16   }
17 }
```

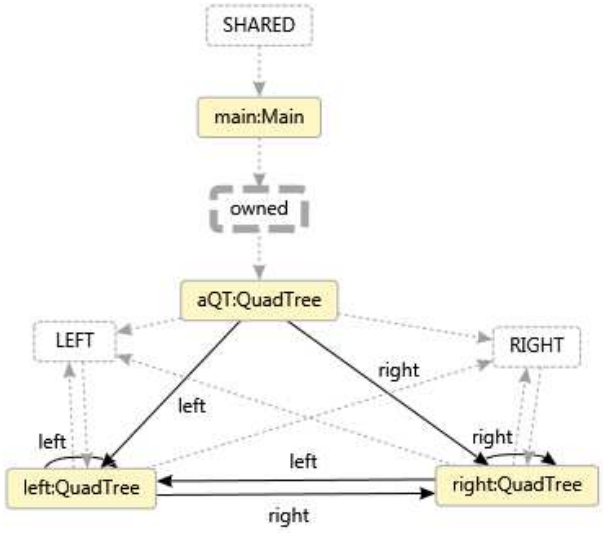


Figure 2.25: Handling recursive types, for a binary tree data structure.

As an example, consider a class `QuadTree`, which declares a field `nwQT` of type `QuadTree` in its `OWNED` private domain (Fig. 2.24). The analysis does four passes over the domain declaration `QuadTree::OWNED`. In the first two passes, the analysis creates two new OObject instances while interpreting the object allocation expressions `newQuadTree` (Fig 2.24 line 1 and line 6) and two new ODomain instances while interpreting the domain declarations (Fig 2.24 line 3 and line 10).

In the third pass, the analysis creates the OObject `QuadTree<main.OWNED.aQT.OWNED>`, but while interpreting the domain declaration `QuadTree::OWNED`, the analysis reuses the existing ODomain `main.OWNED.aQT.OWNED` created during the second pass. By detecting the same ODomain, the analysis can make the OObject `QuadTree<main.OWNED.aQT.OWNED>` to be both the parent and the child of an ODomain, thus creating a cycle. In the last pass, the analysis just ensures the existence of the OObject `QuadTree<main.OWNED.aQT.OWNED>` and the ODomain `main.OWNED.aQT.OWNED`. Since the OGraph remains unchanged, the analysis terminates.

If the class `QuadTree` were to declare two different domains `LEFT` and `RIGHT` (Fig. 2.25, line 10), the analysis would traverse the stack Υ, looking for an existing ODomain that is child of an OObject in the stack and that can be reused. The stack Υ thus ensures the termination of the analysis.

2.9 Extracted Object Graph is Precise

For security, the extraction analysis must not merge objects excessively. For example, an OGraph that represents all the runtime objects with one abstract object or an OGraph that is fully connected is sound but very imprecise. Ideally, an OGraph must have no more OEdges than soundness requires. Like any result of a sound static analysis, however, an OGraph is likely to have false positives and show OObjects or OEdges that do not correspond to runtime objects or runtime relations.

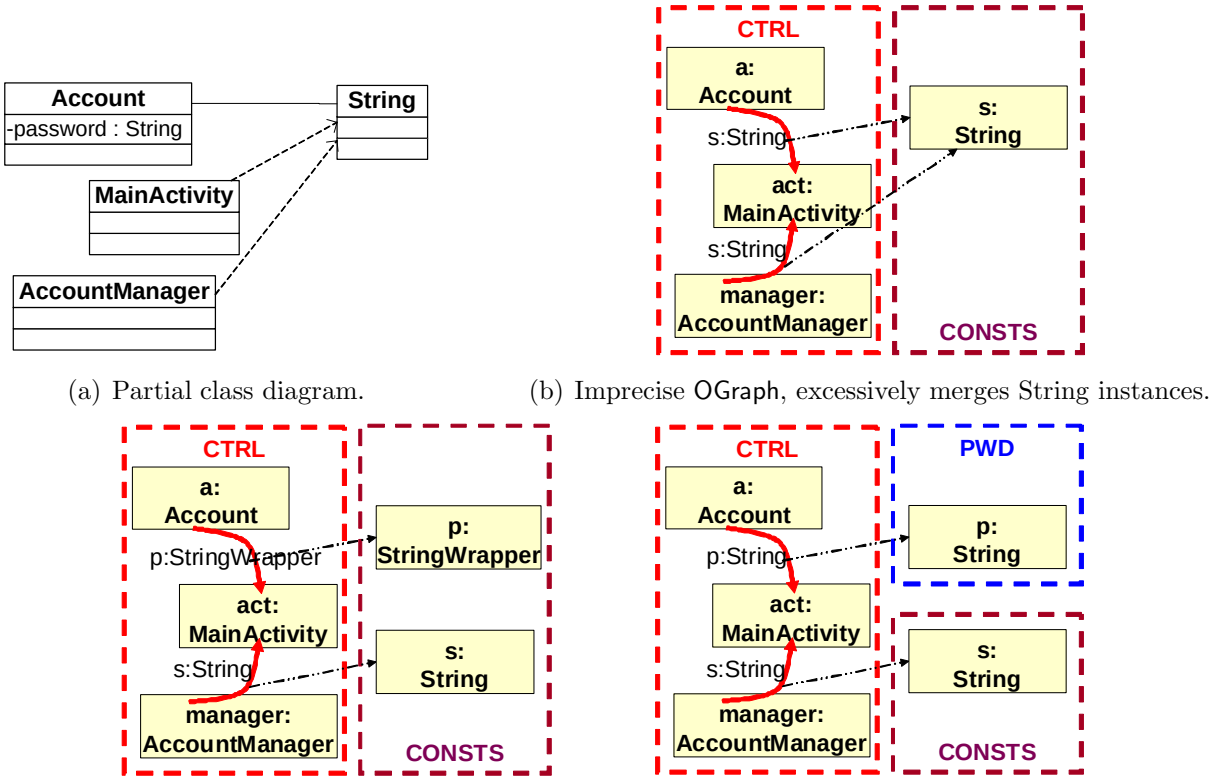
2.9.1 Architects can Fine-Tune Precision with Annotations

To support reasoning about the content of communication, a dataflow edge refers to an abstract object rather than a class. Two edges with the same source and destination can refer to distinct abstract objects that have the same type, but different conceptual meaning. For example, if an application writes in the log a constant of type `String` representing a constant tag and a password, the `OGraph` has two distinct edges, where only one refers to confidential data. The analysis merges objects of type `String` that are in the same domain and both dataflow edges refer to the same object. Then, the architects cannot distinguish between objects representing confidential and non-confidential information. An imprecise object graph may have one representative object `s:String` for all these instances (Fig. 2.26(b)). The precision of such an object graph is similar to the one provided by a class diagram (Fig. 2.26(a)). Architects can create a wrapper class of `String` and the analysis would distinguish between the two objects in the same domain based on their class (Fig. 2.26(c)). However, the later solution implies non-trivial code refactoring. Instead, the architects can refine the annotations and place objects of the same class in different domains (Fig. 2.26(d)).

For a variable in the code, ScoriaX needs to determine the objects on the OGraph that the variable refers to. Without annotations, the analysis would determine the set of objects in the OGraph such that an object in the set has the same type as the variable or a subtype thereof. Instead, by using annotations, the analysis achieves precision and determines only a subset of these objects, i.e., only those objects that are in the domains that correspond to the annotation.

2.9.2 ScoriaX is Flow-Insensitive

A static analysis is flow-sensitive if the conditional statements and the order of the statements in the program affect the result of the analysis [87]. For security, an architect needs



(a) Partial class diagram.

(b) Imprecise OGraph, excessively merges String instances.

(c) More precise OGraph, using a wrapper.

(d) More precise OGraph, by refining annotations.

Figure 2.26: Placing objects of the same type in different domains

an object graph that approximates all possible executions, which include the worst-case scenario. The object graph can be extracted by a flow-insensitive analysis because such a precision seems sufficient for a worst-case approximation. ScoriaX safely assumes that all the classes that have a corresponding new expression in the code are instantiated, and all the methods of these classes can be invoked in an arbitrary order. This assumption is particularly useful for handling callbacks, i.e., methods invoked by a framework, and multi-threaded programs where the order in which statements are executed can be arbitrary. In such cases, a partially flow-sensitive analysis may be unsound [126].

For example in the Android framework, the method `onCreate` of the class `Activity` is invoked every time the user rotates the device (Fig. 2.27). The method `onCreate` discloses the value of the field into an object of type `Log` and, in the last statement, the method assigns confidential data to the field. Therefore, information disclosure occurs only at the second invocation. Similarly, another method, `onResume` assigns confidential value of the field and

```
1  class Main<owner> {
2    domain LOGIC, STATE;
3    MyApp<LOGIC,STATE> app = new MyApp();
4  }
5  class MyApp<owner,S> extends Activity {
6   domain OWNED;
7   TelephonyManager<S> tm = ...;
8   Log<S> log = ...;
9   Datacontainer<OWNED,S> data = new Datacontainer();
10
11  void onCreate(Bundle<owner> b){
12    String<S> imei = tm.getDeviceId(); //source
13    log.i("INFO", data1.value); //sink
14    data.value = imei;
15  }
16  void onResume() {
17     data.value = tm.getDeviceId();
18  }
19 }
20 class Datacontainer{
21   String<S> value = "android";
22 }
```

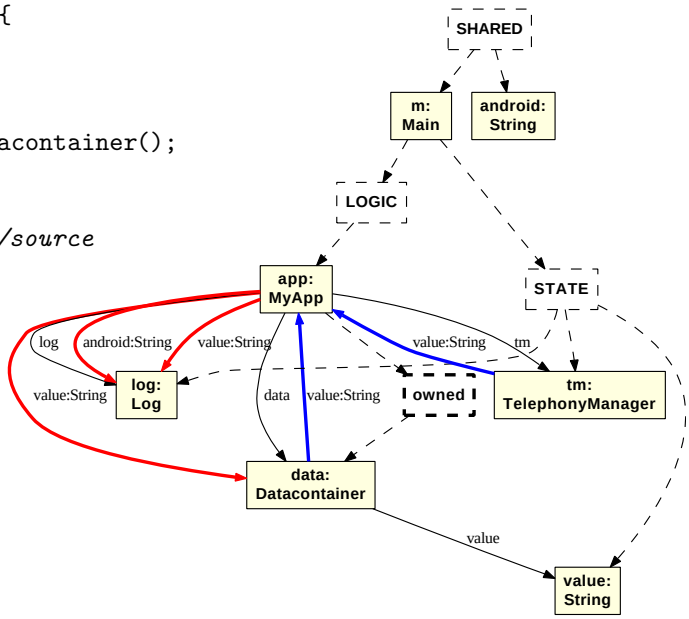


Figure 2.27: The device id flows from an object of type `TelephonyManager` to a an object `value:String` after this object flows to an object of type `Log`. A flow-insensitive approach reports an information disclosure.

the vulnerability exists when events are triggered in a particular order, i.e., the user resumes the application and rotates the device. Without a careful modeling of the lifecycle of objects of type `Activity`, the analysis may miss these security vulnerabilities.

2.9.3 Imprecision of ScoriaX

ScoriaX is a domain-sensitive, object- and flow-insensitive analysis that is over-conservative and favors soundness over precision. Architects can fine-tune the precision of ScoriaX using annotations and use a type checker to ensure that annotations are consistent with each other and with the code. Imprecision may occur in the presence of inheritance and collections.

Imprecision in the presence of inheritance

In the presence of inheritance, the static analysis may introduce false positives. First, the analysis may create false positive objects if a base class has methods that are never invoked. This is due to the over-conservative assumption of ScoriaX that all methods of an instantiated class may be invoked. Other approaches use a pre-computed call graph, but such a graph is challenging to construct in the presence of callbacks, and often requires parsing configuration files. Another imprecision may be due to an invoked method of a super class. For two variables of a general type that are declared in the same domain, the analysis cannot distinguish between objects of the same type that are in the same domain.

Consider for example, two variables `a` and `b` of type `Base` that alias. ScoriaX extracts the abstract objects `m:Main`, `a:A`, `b:B` and `c:B`, and edges from `m:Main` to both `a:A` and `b:B` that refer to `c:C` (Fig. 2.28). Only the method `run` of type `B` is invoked at runtime (line 9), and the dataflow edge from `m:Main` to `a:A` is a false positive. The method `unreachable` (line 14) of the class `Base` is not invoked, however ScoriaX extracts two abstract domains `a.DOM` and `b.DOM`, and two abstract objects `ret:C` in these domains. The objects `ret:C` and the edges from `a:A` and `b:B` to these objects that are false positives.

Imprecision due to flow-insensitivity

ScoriaX is flow-insensitive and safely assumes that statements may be executed in an arbitrary order, which may lead to false positive objects or edges. For the code in Fig. 2.28 (line 7), the `null` value is assigned to the variable `a`. Since the analysis is flow-insensitive, the analysis considers that `a` may-alias the object `a:A` and extracts a false positive dataflow edges from `m:Main` to `a:A` that refers to the abstract object `c:C`.

Imprecision with collections

With respect to collections, ScoriaX does not distinguish between specific elements of a collection, and considers that all the elements are in the same domain. The assumption is

```
1  Main<shared> m = new Main();
2  class Main<owner> {
3   void run(){
4     Base<ADOM> a = new A<ADOM>();
5     Base<ADOM> b = new B<ADOM>();
6     C<ADOM> c = new C<ADOM>();
7     a = null;
8     a = b;
9     a.run(c);
10   }
11 class Base<owner>{
12    domain DOM;
13    void run(C<owner> c){ ... }
14    void unreachable(){
15      C<DOM> ret = new C<DOM>
16      ret.mc("const");
17    }
18 }
19 class C {
20    void mc(String<shared> s){ ... }
21 }
```

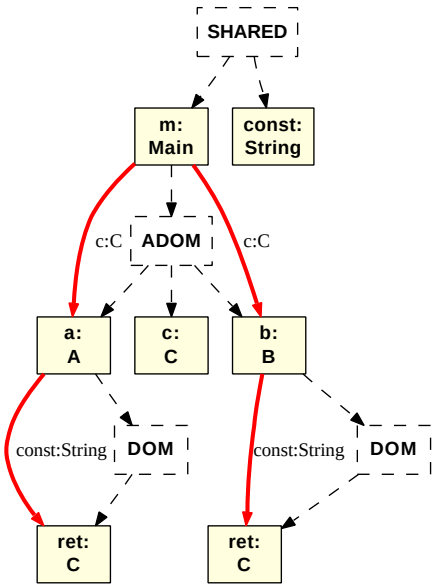


Figure 2.28: Imprecision due to the invocation of a method in a super class and unreachable code.

common in static analysis research [39, 48] and may lead to imprecision when a collection has both confidential and non confidential information. A collection has domain parameter to specify the domain of its elements, but all the elements are in the same domain (Fig. 2.5).

2.10 Scalability of ScoriaX

Traditionally, scalability of a static analysis is reported to the size of the program that is measured in lines of code. However, to discuss the scalability of ScoriaX one needs to consider not only the lines of code, but also the number of object allocation expressions that the code has. ScoriaX uses abstract interpretation and traverses the code starting from a root class specified by the architects. The analysis then traverses each class declaration that is instantiated in the code multiple times.

In practice, classes that belong to libraries and frameworks are often instantiated multiple times. To improve scalability, ScoriaX accepts summaries of these library classes as

input where each summary includes a class declaration with field and method signatures without the method bodies. Although incomplete summaries may introduce unsoundness, summaries have advantages besides scalability. For example, the library code may not always be available to be annotated, and often the application uses only a small part of the library.

The main bottleneck in using ScoriaX on large systems is the process of adding annotations, which is manual with limited aid of tools that provide boilerplate annotations. Related work measured the effort of adding annotations to be 1hour/KLOC [6]. Since annotations leverage a type system, they are amenable to automatic inference [64] that would reduce the effort and improve scalability. Inference works well with for strictly encapsulated objects in private domains, but not for objects in public domains.

2.11 Summary

This chapter informally describes a static analysis that extracts an approximation of the runtime architecture as a sound, hierarchical object graph with dataflow edges that refer to object. Achieving soundness (*H1c*) of the abstract graph requires a static analysis, but a hierarchical organization of objects (*H1a*) is unavailable in code written in mainstream programming languages. To support legacy code (*H1b*), the static analysis uses annotations that are consistent with each other and with the code. Chapter 3 describes the analysis formally, proves the soundness theorems and discusses how by design the analysis considers aliasing (*H1d*) and supports summarization(*H1f*). Chapter 4 evaluates the implementation of the analysis focusing on achieving precision (*H1e*) for reasoning about security and discusses the scalability (*H1h*) of the analysis.

Chapter 3 Formalization of Extraction

To show that ScoriaX meet the extraction requirement (*H1c*), this chapter[1] formalizes the analysis and proves object soundness and edge soundness. The analysis considers aliasing (*H1d*) using the Ownership Domains type system [10] and is domain-sensitive. The analysis leverages Ownership Domains annotations in order to extract a hierarchical organization of objects (*H1a*), where an object can have several domains.

Extensions of Ownership Domains allow an object to be borrowed from one domain to another, or to be passed linearly between domains without keeping a persistent reference to it. To support extensions of Ownership Domains, the analysis keeps track of the value flow and attempts to resolve from which domain an object is borrowed, and what is the domain to which an object is passed linearly.

I split the formalization in two parts. In the first part, Section 3.1 formalizes the extraction analysis on Ownership Domains and, Section 3.2 introduces the soundness theorems. In the second part, Section 3.3 focuses on the value flow analysis. Since the value flow analysis and the extraction analysis are inter-dependent, some repetitions in the formalization occur, but the explanations focus on the value flow analysis. Next, Section 3.4 discusses the implementation of the analysis and extraction of other types of edges such as creation edges and control flow edges. Finally, Section 3.5 discusses differences with previous versions of the analysis and Section 3.6 concludes.

3.1 Formalization on Ownership Domains

The formalization is based on a Java-like program semantics with Ownership Domain annotations. Section 3.1.1 presents the syntax of the language used in the formalization. Section 3.1.2 defines the data types of the OGraph. Section 3.1.3 formalizes the extraction

[1] Parts of the formalization appeared in the informal proceedings of FOOL13 [132].

analysis. Section 3.1.4 presents instrumentation of dynamic semantics, which are required by the soundness proof.

3.1.1 Abstract Syntax

I formalize the extraction static analysis on Featherweight Domain Java (FDJ), which models a core of the Java language with Ownership Domains [10]. To keep the language simple and easier to reason about, FDJ ignores Java language constructs such as interfaces and static code. From the FDJ abstract syntax, I exclude cast expressions and domain links, which are part of FDJ, but not crucial to our discussion (Fig. 3.1).

In FDJ, C ranges over class names; T ranges over types; f ranges over field names; v ranges over values; d ranges over domain names; e ranges over expressions; x ranges over variable names; n ranges over values and variable names; S ranges over stores; ℓ and θ range over locations in a store; θ represents the value of `this`; a store S maps locations ℓ to their contents; the set of variables includes the distinguished variable `this` of type T_{this} used to refer to the receiver of a method; the result of the computation is a location ℓ, which is sometimes referred to as a value v; $S[\ell]$ denotes the store entry of ℓ; $S[\ell, i]$ denotes the value of i^{th} field of $S[\ell]$; $S[\ell \mapsto C\!<\!\overline{\ell'.d}\!>\!(\overline{v})]$ denotes adding an entry for location ℓ to S; α and β range over formal domain parameters; m ranges over method names; p ranges over formal domain parameters, actual domains, or the special domain `SHARED`; the expression form $\ell \triangleright e$ represents a method body e executing with a receiver ℓ; an overbar denotes a sequence; the fixed class table CT maps classes to their definitions; a program is a tuple (CT, e) of a class table and an expression; Γ is the typing context; and Σ is the store typing.

3.1.2 Object Graph

The analysis extracts a hierarchical object graph (OGraph) with nodes that represent abstract objects (OObjects) and group of objects (ODomains), and edges that represent dataflow communication between abstract objects (Fig. 3.10). The OGraph is a triplet $G =$

$$
\begin{array}{rcll}
CT & ::= & \overline{cdef} & \textit{class table} \\
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} & \\
 & & \{\ \overline{dom};\ \ \overline{T}\ \overline{f};\ \ C(\overline{T'}\ \overline{f'},\overline{T}\ \overline{f}) & \\
 & & \{\texttt{super}(f');\texttt{this}.\overline{f}=\overline{f};\}\ \ \overline{md}\ \} & \textit{class declaration} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; & \textit{domain declaration} \\
md & ::= & T_R\ m(\overline{T}\ \overline{x})\ \ T_{this}\ \{\texttt{return}\ e_R;\} & \textit{method declaration} \\
e & ::= & x\ \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e})\ \mid e.f\ \mid e.f=e' & \\
 & & \mid e.m(\overline{e})\ \mid \ell\ \mid \ell \triangleright e & \textit{expressions} \\
n & ::= & x\ \mid\ v & \textit{values or variable names} \\
p & ::= & \alpha\ \mid\ n.d\ \mid\ \texttt{SHARED} & \textit{domain name} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} & \textit{type: class and list of domains} \\
v,\ell,\theta & \in & \textit{locations} & \\
S & ::= & \ell\ \rightarrow\ C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) & \textit{location store} \\
\Sigma & ::= & \ell\ \rightarrow\ T & \textit{store typing} \\
\Gamma & ::= & x\ \rightarrow\ T & \textit{type environment}
\end{array}
$$

Figure 3.1: Simplified FDJ abstract syntax [10]. For readability, repeated from Fig. 2.9.

$\langle DO, DD, DE\rangle$, where DO is a set of OObjects, and DE is a set of OEdges. Each OEdge is a directed edge from a source O_{src} to a destination O_{dst}. The label of an OEdge is the OObject that the dataflow refers to. The flag distinguishes between OEdges that represent import and export dataflow communication. The OGraph is a multi-graph, where multiple edges that refer to different OObjects might exists between the same source and destination. To construct the object-domain hierarchy, the analysis uses DD which maps a pair $(O, C{::}d)$ to an ODomain D and is also used to keep track of domain parameters. To ensure termination, the analysis uses the stack Υ, that records combination of class and actual domains $C\texttt{<}\overline{D}\texttt{>}$ analyzed.

ScoriaX meets the aliasing requirement and relies on the precision about aliasing that Ownership Domains offer, and avoids merging object excessively. The Ownership Domains type system guarantees that two objects in different domains cannot alias. The analysis only merges two objects of the same class if all their domains are the same. The analysis is domain-sensitive and distinguishes between different instances of the same class C that are in different domains, even if created at the same `new` expression in the program. In addition, the analysis treats an instance of class C with actual parameters $\overline{p}$ differently from another instance that has actual parameters $\overline{p'}$. Hence, the data type of an OObject uses $C\texttt{<}\overline{D}\texttt{>}$.

$$
\begin{array}{ll}
G \in \mathsf{OGraph} & ::= \langle\ \mathbf{Objects} = DO,\ \mathbf{DomainMap} = DD, \mathbf{Edges} = DE\ \rangle \\
D \in \mathsf{ODomain} & ::= \langle\ \mathbf{Id} = D_{id},\ \mathbf{Domain} = C{::}d\ \rangle \\
O \in \mathsf{OObject} & ::= \langle\ \mathbf{Type} = C{<}\overline{D}{>}\ \rangle \\
E \in \mathsf{OEdge} & ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Label} = O_{label}, \mathbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle \\
DD & ::= \emptyset\ \mid\ DD \cup \{\ (O, C{::}d) \mapsto D\ \} \cup \{\ (O, C{::}\alpha) \mapsto D\ \} \\
DO & ::= \emptyset\ \mid\ DO \cup \{\ O\ \} \\
DE & ::= \emptyset\ \mid\ DE \cup \{\ E\ \} \\
\Upsilon & ::= \emptyset\ \mid\ \Upsilon \cup \{\ C{<}\overline{D}{>}\ \} \\
H & ::= \emptyset\ \mid\ H \cup \{\ \ell \mapsto O\ \} \\
K & ::= \emptyset\ \mid\ K \cup \{\ \ell.d \mapsto D\ \} \\
L_I & ::= \emptyset\ \mid\ L_I \cup \{\ (\ell_{src}, \ell_{dst}) \mapsto \{E\}\} \\
L_E & ::= \emptyset\ \mid\ L_E \cup \{\ (\ell_{src}, \ell_{dst}) \mapsto \{E\}\}
\end{array}
$$

Figure 3.2: Data type declarations for the OGraph.

The formalization follows the FDJ convention and considers an OObject's owning ODomain as the first element D_1 of $\overline{D}$.

The OGraph is hierarchical and has implicit ownership edges in addition to the OEdges that have source and destination OObjects. The OGraph is also well-formed with respect to the ownership relations declared in the code using the annotations. The data type declaration of the OGraph implicitly captures the object-domain hierarchy using the DD map without defining directly a set of ownership edges. An ownership edge states that an OObject $O = \langle C{<}\overline{D}{>}\rangle$ is a child of D_1, or that O owns a domain D. Given a mapping $\{(O, C'{::}d) \mapsto D\}$ in DD where $C'{::}d$ is a domain declaration, D is a child of O. Since domains are inherited from a base class to a subclass [10], the class C of O can be a subclass of C' where d is declared. The analysis also uses DD to map formal domain parameters $C{::}\alpha$ to actual domains.

Although a domain d is declared by a class C, each runtime instance of type C gets its own runtime domain $\ell.d$. For example, if there are two distinct object locations ℓ and ℓ' of class C, then $\ell.d$ and $\ell'.d$ are distinct. Since an ODomain represents a runtime domain $\ell_i.d_i$, one domain declaration d in the code can create multiple ODomains D_i in the OGraph and the fresh identifier D_{id} ensures that multiple ODomains can be created for the same

domain declaration C::d. Since no class declares the `SHARED` domain, the analysis qualifies it as ::`SHARED`.

The analysis starts by creating a global ODomain $D_{\texttt{shared}}$, the root of the OGraph. A developer picks a root class, C_{root}, and the analysis creates O_{root} in $D_{\texttt{shared}}$. The analysis also requires an initial context that is a dummy OObject O_{world}, which does not correspond to an actual runtime object. Next, the analysis changes the context from O_{world} to O_{root}, and analyzes recursively all the expressions in the methods of C_{root}.

Instrumentation. Proving soundness means proving that there is a map between a Runtime Object Graph (ROG) and an OGraph by instrumenting the dynamic semantics (Fig. 3.10) using the maps H, K, L_I, and L_E. H maps a location ℓ to the corresponding OObject, and K maps a runtime domain $\ell.d$ to an ODomain. The multi-valued maps L_I and L_E map a pair of locations (ℓ_{src}, ℓ_{dst}) to a set of OEdges $\{E\}$. We use two maps for edges because a pair ($H[\ell_1]$, $H[\ell_2]$) can be associated with an import edge from $H[\ell_1]$ to $H[\ell_2]$, or with an export edge from $H[\ell_1]$ to $H[\ell_2]$.

Notation. For a map M, a key k, and a value v, we use $M[k]$ to denote the lookup of k, and $M' = M[k \mapsto v]$ for adding an entry for k to M. For a multi-valued map M, we use the notation $M' = M[k \mapsto_{\cup} \{v\}]$ for adding an entry for k to M. If the map already has an entry for k, the resulting value is the union of the existing value set and $\{v\}$.

3.1.3 Static Semantics

The formalization uses a constraint-based specification as a set of inference rules. In FDJ, a program is a tuple (CT, e) that consists of a class table CT, which maps classes to their definitions, and an expression e. The analysis starts with a root expression e_{root}, that explicitly instantiates the root class C_{root}. The analysis result is the least solution $G = \langle DO, DD, DE \rangle$ of the following constraint system:

$$\emptyset, \emptyset, G \vdash (CT, e_{root})$$

The analysis starts by creating the OObject O_{world} and its owning ODomain $D_{\texttt{SHARED}}$, which constitutes the root of the OGraph,

$$D_{\texttt{SHARED}} = \langle D_0, ::\texttt{SHARED} \rangle \qquad\qquad O_{world} = \langle C_{dummy} < . > \rangle$$

then abstractly interprets e_{root} in the context of O_{world}:

$$\emptyset, \emptyset, G \vdash_{O_{world}} e_{root}$$

The judgement form for expressions is as follows:

$$\Gamma, \Upsilon, G \vdash_{O,H} e$$

The O subscript on the turnstile captures the context-sensitivity, and represents the context object that the analysis uses to abstractly interpret e. Since an OObject O is uniquely identified based on its domains, the context is a domain and the analysis is domain-sensitive. The H subscript is a map used by the dynamic semantics and the store typing rule in the static semantics (not shown). For readability, we omit H when not in use. $CT(C)$ and $CT(\texttt{Object})$ represent a lookup of a class C and the class `Object` in the class table[2], and is an implicit clause in all the static rules.

In Df-New, the analysis interprets an object allocation in the context of O and ensures the analysis creates objects and domains. The analysis first ensures that DO contains an OObject O_C for the newly allocated object. Then, using *dparams*, Df-New ensures that each of the actual domain parameters p_i maps to an actual domain D_i in the context of O, where the corresponding formal domain parameter α_i maps to the same D_i but in the context of O_C. Df-New also ensures that the object hierarchy is created such that new ODomains are created for each domain declarations in C according to the auxiliary judgment *ddomains*.

[2]These clauses are listed once at the top of Fig. 3.3 to avoid repetition.

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$$

$$CT(\texttt{Object}) = \texttt{class Object<}\alpha_o\texttt{>}\ \ \{\ \ \}$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad O = C_{\texttt{this}}\texttt{<}\overline{D_O}\texttt{>} \qquad \forall i \in 1..|\overline{p}| \qquad G \vdash_O D_i \in findD(C_{\texttt{this}}::p_i) \\ O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle \qquad \{O_C\} \subseteq DO \\ G \vdash_O dparams(C, O_C) \qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD \\ \Upsilon, G \vdash_O ddomains(C, O_C) \\ \forall m \in \overline{md}\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \implies \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, G \vdash_{O_C} e_R \\ \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e})}\ [\textsc{Df-New}]$$

$$\frac{\begin{array}{c} e_0 : C\texttt{<}\overline{p}\texttt{>} \qquad (T_k\ f_k) \in fieldDecls(C) \\ G \vdash_O import(C\texttt{<}\overline{p}\texttt{>}, T_k) \\ \Gamma, \Upsilon, G \vdash_O e_0 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k}\ [\textsc{Df-Read}] \qquad \frac{\begin{array}{c} e_0 : C\texttt{<}\overline{p}\texttt{>} \qquad (T_k\ f_k) \in fields(C\texttt{<}\overline{p}\texttt{>}) \\ e_1 : C_1\texttt{<}\overline{p''}\texttt{>} \qquad C_1\texttt{<}\overline{p''}\texttt{>} <: T_k \\ G \vdash_O export(C\texttt{<}\overline{p}\texttt{>}, C_1\texttt{<}\overline{p''}\texttt{>}) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O e_1 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k = e_1}\ [\textsc{Df-Write}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \\ G \vdash_O O_i \in lookup\ (T_{src}) \\ G \vdash_{O_i} O_j \in lookup\ (T_{label}) \\ \{\langle O_i, O, O_j, Imp\rangle\} \subseteq DE \end{array}}{G \vdash_O import\ (T_{src}, T_{label})}\ [\textsc{Aux-Import}] \qquad \frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \\ G \vdash_O O_i \in lookup\ (T_{dst}) \\ G \vdash_O O_j \in lookup\ (T_{label}) \\ \{\langle O, O_i, O_j, Exp\rangle\} \subseteq DE \end{array}}{G \vdash_O export\ (T_{dst}, T_{label})}\ [\textsc{Aux-Export}]$$

$$\frac{\begin{array}{c} e_0 : C\texttt{<}\overline{p}\texttt{>} \qquad mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T'} \to T'_R \qquad mtypeDecl(m, C) = \overline{T_f} \to T_R \\ G \vdash_O import(C\texttt{<}\overline{p}\texttt{>}, T_R) \\ \forall k \in 1..|\overline{e}|\ e_k : T_a \qquad T_a <: T'_k \qquad G \vdash_O export(C\texttt{<}\overline{p}\texttt{>}, T_a) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.m(\overline{e})}\ [\textsc{Df-Invk}]$$

Figure 3.3: Static semantics.

Both *dparams* and *ddomains* are recursive auxiliary judgments that consider inheritance, i.e., the domain may be declared by a class C' that C extends. The base case for the recursion is the `java.lang.Object` class (Fig. 3.3).

The analysis terminates in the presence of recursive types. Df-New uses the auxiliary judgement Aux-Dom to ensure that DD has an ODomain corresponding to each domain that C locally declares. If C is a recursive type, *ddomains* first checks if Υ already contains the combination $C\texttt{<}\overline{D}\texttt{>}$. If such a combination exists, the analysis reuses the ODomain D_{rec} that the key $(\langle C\texttt{<}\overline{D}\texttt{>}\rangle, C::d_j)$ maps to in DD and adds a cycle in the object-domain hierarchy by mapping $(O_C, C::d_j)$ to D_{rec}. Otherwise, it creates a fresh ODomain D_j and

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad O_C = C\text{<}\overline{D}\text{>} \\ \forall \alpha_j \in params(C)\ \{(O_C, C{::}\alpha_j) \mapsto D_j\} \subseteq DD \\ G \vdash_O dparams(C', O_C) \end{array}}{G \vdash_O dparams(C, O_C)}[\textsc{Aux-Alpha}]$$

$$\frac{CT(\texttt{Object}) = \texttt{class Object}\text{<}\alpha_o\text{>}\ \{\ \}}{G \vdash_O dparams(\texttt{Object}, O_C)}[\textsc{Aux-Alpha1}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad O = C\text{<}\overline{D_O}\text{>} \\ n : C_n\text{<}\overline{p}\text{>} \qquad G \vdash_O O_i \in lookup\ (C_n\text{<}\overline{p}\text{>}) \\ D_i = DD[(O_i, C_n{::}d)] \end{array}}{G \vdash_O D_i \in findD\ (C{::}n.d)}[\textsc{Aux-Find-Public}]$$

$$\frac{G = \langle DO, DD, DE\rangle \qquad O = C\text{<}\overline{D_O}\text{>} \qquad D_i = DD[(O, C{::}d_i)]}{G \vdash_O D_i \in findD\ (C{::}\texttt{this}.d_i)}[\textsc{Aux-FindThis}]$$

$$\frac{G = \langle DO, DD, DE\rangle \qquad O = C\text{<}\overline{D_O}\text{>} \qquad D_i = DD[(O, C{::}\alpha_i)]}{G \vdash_O D_i \in findD\ (C{::}\alpha_i)}[\textsc{Aux-FindD}]$$

$$\frac{}{G \vdash_O D_{\texttt{SHARED}} \in findD\ (::\texttt{shared})}[\textsc{Aux-FindShared}]$$

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\text{<}\overline{\alpha}, \overline{\beta}\text{>}\ \texttt{extends}\ C'\text{<}\overline{\alpha}\text{>}\ \{\ldots\ \overline{dom};\ \ldots;\ \} \qquad G = \langle DO, DD, DE\rangle \\ \forall(\texttt{domain}\ d_j) \in \overline{dom} \qquad O'_C = \langle C''\text{<}\overline{D''}\text{>}\rangle \qquad O_C = \langle C\text{<}\overline{D}\text{>}\rangle \\ \exists C''\text{<}\overline{D''}\text{>} \in \Upsilon \cup \{C\text{<}\overline{D}\text{>}\}\ \text{s.t.}\ C'' = C \wedge D_{rec} = DD[(O'_C, C{::}d_j)] \Longrightarrow \{(O_C, C{::}d_j) \mapsto D_{rec}\} \subseteq DD \\ \forall C''\text{<}\overline{D''}\text{>} \in \Upsilon \cup \{C\text{<}\overline{D}\text{>}\}\ \text{s.t.}\ C'' \neq C \Longrightarrow (D_j = \langle D_{id_j},\ C{::}d_j\rangle \qquad \{(O_C, C{::}d_j) \mapsto D_j\} \subseteq DD) \\ \Upsilon, G \vdash_O ddomains(C', O_C) \end{array}}{\Upsilon, G \vdash_O ddomains(C, O_C)}[\textsc{Aux-Dom}]$$

$$\frac{}{\Upsilon, G \vdash_O ddomains(\texttt{Object}, O_C)}[\textsc{Aux-Obj1}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \\ O = C_{\texttt{this}}\text{<}\overline{D}\text{>} \qquad O_k \in DO \qquad O_k = \langle C\text{<}\overline{D}\text{>}\ \rangle \qquad C <: C' \\ \forall i \in 1..|\overline{p'}| \qquad G \vdash_O D'_i \in findD(C_{\texttt{this}}{::}p'_i) \qquad D'_i = D_i \end{array}}{G \vdash_O O_k \in lookup\ (C'\text{<}\overline{p'}\text{>})}[\textsc{Aux-Lookup}]$$

Figure 3.4: Auxiliary judgments for static semantics.

maps $(O_C, C{::}d_j)$ to D_j. Aux-Dom recursively includes inherited domains from base classes as well. Aux-Obj1, the base case of the recursion, deals with the class `Object`, for which Aux-Obj1 does nothing, because `Object` has no fields, domains, or methods in FDJ.

The analysis assumes that all the methods of class C may be invoked. Df-New obtains each return expression e_R in each method body m of C, and recursively processes e_R in the context of the new OObject O_C. To avoid infinite recursion, before Df-New analyzes e_R, it

checks if the combination of the class C and actual domains $\overline{D}$ have been previously analyzed by looking for this combination in Υ. If this combination does not exist, Df-New pushes on Υ the current combination. As a side note, Υ tracks previously created OObjects that lead to the creation of O_C as the call stack of the analysis. It does not do so globally across the program because similar combinations of the same class and domain parameters can occur in different contexts, and must be analyzed separately. Finally, Df-New analyzes each argument of the constructor. Since the analysis distinguishes between a field initialization in a constructor and a field write, Df-New does not require dataflow edges in DE.

The auxiliary judgement *lookup* returns the set of the OObjects O_k in DO such that the class of O_k is C' or one of its subclasses. It also ensures that each domain D_i of O_k corresponds to D'_i, a domain associated with O in DD. Due to subtyping, the number of actual domain parameters $\overline{p}$ is smaller than or equal to the number of actual ODomains $\overline{D}$. This is how the analysis achieves precision, because *lookup* returns only a subset of all the objects of class C' or its subclasses in DO. From this subset, the analysis picks the source or destination OObjects, and finds the flow object of an OEdge.

The auxiliary judgement Aux-Export ensures that export edges exist between the context OObject O and each of the OObjects O_i that *lookup* (T_{src}) returns. The auxiliary judgment *lookup* invoked using the T_{label} argument returns the set of OObjects O_j that the dataflow edges refer to. As a result, there could be multiple edges that refer to different `OObjects` between the same source and destination. Aux-Import is similar to Aux-Export, but the edge has an opposite direction from O_i to the context O. Another difference is that Aux-Import invokes the second *lookup* in the context of O_i rather than O because the imported object exists in the context of the receiver O_i.

Df-Read and Df-Write abstractly interpret field read and field write expressions, and use Aux-Import and Aux-Export, respectively. Both auxiliary judgements take the type e_0 as the first argument, and pass it to *lookup* to set the source and destination OObjects. For the OObject that the dataflow edge refers to, Df-Read uses the type of the field f_k,

$$\frac{}{\Gamma, \Upsilon, G \vdash_O x}[\textsc{Df-Var}] \qquad \frac{}{\Gamma, \Upsilon, G \vdash_O \ell}[\textsc{Df-Loc}] \qquad \frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, G \vdash_{O_C} e}{\Gamma, \Upsilon, G \vdash_{O,H} \ell \triangleright e}[\textsc{Df-Context}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE \rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

Figure 3.5: Static semantics (continued).

while Df-Write uses the type of the right-hand side expression e_1.

Df-Invk abstractly interprets method invocation expressions. First, it ensures the existence of import edges from the receiver of the method to the context OObject O. These import edges refer to the OObjects that the analysis finds by using *lookup* on the return type of the method in the context of the receiver. Next, for each argument e_k, Df-Invk ensures the existence of the export edges from O to the receiver of the method that refer to the OObject the analysis finds by using *lookup* on the type of each argument. The rule ensures export edges only for a method invocation with at least one argument.

Df-Var, Df-Loc, and the rest of the rules complete our formalization and make the induction go through (Fig. 3.5). Df-Context analyzes expressions of the form $\ell \triangleright e$. The context for analyzing e changes from O to O_C, where O_C is the result of looking up the receiver ℓ in H. Finally, the induction requires an augmented store typing rule, Df-Sigma, to ensures that the method bodies have been analyzed for all the locations ℓ in the store, and that every ℓ has a corresponding OObject in DO. To denote all the objects in the store, the rule uses the subscript CT instead of O.

Figure 3.6 shows the definitions the analysis uses to qualify a domain p by the class C that declares it. In the context of Γ, Σ, and θ, Qual-Var qualifies $n.d$ as $C{::}d$. This judgement also applies to the case when n is `this` and $p = \texttt{this}.d$. Qual-Param qualifies a formal domain parameter α as $C{::}\alpha$, where C is the class of `this`. Since no class declares the `shared` domain, Qual-Shared qualifies it as `::shared`. We use these rules implicitly in the static and dynamic semantics to ensure that $(O, C{::}d) \mapsto D$ is in DD.

$$\frac{\Gamma;\Sigma;\theta \vdash n : C\langle\overline{p'}\rangle \qquad d \in domains(C\langle\overline{p'}\rangle)}{\Gamma;\Sigma;\theta \vdash qual(n.d) = C::d}[\text{Qual-Var}]$$

$$\frac{\Gamma;\Sigma;\theta \vdash \texttt{this} : C_{this}\langle\overline{p'}\rangle \qquad \alpha \in params(C_{this})}{\Gamma;\Sigma;\theta \vdash qual(\alpha) = C_{this}::\alpha}[\text{Qual-Param}]$$

$$\frac{}{\Gamma;\Sigma;\theta \vdash qual(\texttt{shared}) = ::\texttt{shared}}[\text{Qual-Shared}]$$

Figure 3.6: Qualify domains rules.

3.1.4 Dynamic Semantics

To complete the formalization, we instrumented the dynamic semantics (Fig. 3.7). The instrumentation extends the dynamic semantics of FDJ [10] (the common parts are highlighted), but is safe since discarding it produces exactly the FDJ dynamic semantics. The instrumented evaluation rule is of the following form:

$$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$$

where $G = \langle DO, DD, DE\rangle$ is the statically computed object graph, and $\rightsquigarrow_G$ means that the expression e evaluates to e' in the context of θ, the value of `this`. The rules in the dynamic semantics keep G unchanged, but change the store S and the maps H, K, L_I, and L_E.

IR-New adds a new location ℓ to the store S, where ℓ maps to an object of type C with the specified ownership domain parameters, and the fields set to the values $\overline{v}$ passed to the constructor. The rule extends H by mapping ℓ and the OObject O_C from DO. The rule requires that each actual domains p_i passed during instantiation has a corresponding actual domain D_i of O_C. Next, the rule extends K such that for all the domains $C::d_j$, the pair $(O_C, C::d_j)$ has a corresponding D_j in DD.

IR-Read and IR-Write ensure that an OEdge E exists between the context OObject O and the receiver O_ℓ. They use θ and ℓ to lookup these OObjects in H. They also ensure that the OObject O_v is of a subclass of the field class C_i as returned by the auxiliary judgment

$$\frac{\boxed{\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\!<\!\overline{p}\!>\!(\overline{v})]} \quad G = \langle DO, DD, DE\rangle \quad \overline{p} = \overline{\ell'.d} \quad \forall i \in 1..|\overline{\ell'.d}|\ D_i = K[\ell'_i.d_i] \quad \ell_i \in dom(H) \text{ s.t. } H[\ell_i] = O_i \quad D_i = DD[O_i, qual(\ell'_i.d_i)] \quad O_C = \langle C\!<\!\overline{D}\!>\rangle \quad O_C \in DO \quad H' = H[\ell \mapsto O_C] \quad \forall(\texttt{domain}\ d_j) \in domains(C\!<\!\overline{p}\!>) \quad D_j = DD[(O_C, C{::}d_j)] \quad K' = K[\ell.d_j \mapsto D_j]}{\theta \vdash \boxed{\texttt{new}\ C\!<\!\overline{p}\!>\!(\overline{v}); S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{\ell; S'}; H'; K'; L_I; L_E}\text{[IR-New]}$$

$$\frac{\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad fields(C\!<\!\overline{p}\!>) = \overline{T}\ \overline{f}} \quad O = H[\theta] \quad O_\ell = H[\ell] \quad O_v = H[v_i] \quad T_i \in \overline{T} \quad E = \langle O_\ell, O, O_v, Imp\rangle \in DE \quad H; K; L_I; L_E \vdash O_v \in irLookup(T_i) \quad L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E\}]}{\theta \vdash \boxed{\ell.f_i; S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{v_i; S}; H; K; L'_I; L_E}\text{[IR-Read]}$$

$$\frac{\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad fields(C\!<\!\overline{p}\!>) = \overline{T}\ \overline{f}} \quad \boxed{S' = S[\ell \mapsto C\!<\!\overline{p}\!>\!([v/v_i]\overline{v})]} \quad O = H[\theta] \quad O_\ell = H[\ell] \quad O_v = H[v] \quad H; K; L_I; L_E \vdash O_v \in irLookup(T_i) \quad T_i \in \overline{T} \quad E = \langle O, O_\ell, O_v, Exp\rangle \in DE \quad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E\}]}{\theta \vdash \boxed{\ell.f_i = v; S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{v; S'}; H; K; L_I; L'_E}\text{[IR-Write]}$$

$$\frac{\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad mbody(m, C\!<\!\overline{p}\!>) = (\overline{x}, e_R)} \quad O = H[\theta] \quad O_\ell = H[\ell] \quad mtype(m, C\!<\!\overline{p}\!>) = \overline{T} \to T_R \quad H; K; L_I; L_E \vdash O_r \in irLookup(T_R) \quad E' = \langle O_\ell, O, O_r, Imp\rangle \in DE \quad L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}] \quad \forall k \in 1..|\overline{x}|\ O_k = H[v_k] \quad H; K; L_I; L_E \vdash O_k \in irLookup(T_k) \quad T_k \in \overline{T} \quad E_k = \langle O, O_\ell, O_k, Exp\rangle \in DE \quad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E_k\}]}{\theta \vdash \boxed{\ell.m(\overline{v}); S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{\ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S}; H; K; L'_I; L'_E}\text{[IR-Invk]}$$

$$\frac{}{\theta \vdash \boxed{\ell \triangleright v; S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{v;\ S}; H; K; L_I; L_E}\text{[IR-Context]}$$

$$\frac{O_k \in rng(H) \quad O_k = \langle C'\!<\!\overline{D'}\!>\rangle \quad C' <: C \quad \forall i \in 1..|\overline{\ell'.d}|\ D_i = K[\ell'_i.d_i] \quad D'_i = D_i}{H; K; L_I; L_E \vdash O_k \in irLookup(C\!<\!\overline{\ell'.d}\!>)}\text{[IR-Lookup]}$$

Figure 3.7: Instrumented dynamic semantics (core rules).

irLookup. Finally, the rules extend the maps L_I and L_E, respectively, by adding E to the set of edges associated with (ℓ, θ) in L_I, and (θ, ℓ) in L_E.

IR-Invk ensures that an import OEdge E' exists from the receiver O_ℓ to the context O, having as the edge's label an OObject of a subclass of the return class C_R. IR-Invk also ensures that an export OEdge E_k exist from O to O_ℓ for every parameter and refers to an OObject of a subclass of the method's parameter class C_k. The rule uses θ and ℓ to lookup

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-New}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.f_i; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Read}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.f_i = e_1; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Write-Rcv}]$$

$$\frac{\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ v.f_i = e_1'; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Write-Arg}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.m(\overline{e}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-RecvInvk}]$$

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ v.m(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-ArgInvk}]$$

$$\frac{\ell \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L_I'; L_E'}{\theta \vdash \ell \triangleright e; S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright e'; S'; H'; K'; L_I'; L_E'}[\text{IRC-Context}]$$

Figure 3.8: Instrumented dynamic semantics (congruence rules).

O and O_ℓ in H. It extends both L_I and L_E by adding E' to the set of import edges between the locations ℓ and θ in L_I, and by adding each E_k to the set of export edges between the locations θ and ℓ in L_E.

The IR-Lookup auxiliary judgment returns the set of OObject found by looking up each actual domain $\ell_i'.d_i$ in K. When the method return expression reduces to a value v, IR-Context propagates v outside of its method context. This rule does not affect the execution of the program.

Finally, the dynamic semantics has the same congruence rules as FDJ [10] (Fig. 3.8). In addition, there are two congruence rules for field-write: IRC-Write-Rcv and IRC-Write-Arg. IRC-Write-Rcv states that the receiver expression e_0 reduces to e_0', while IRC-Write-Arg states that the right-hand side expression e_1 reduces to e_1'.

3.2 Soundness

Section 2.6 (page 21) informally describes soundness of the extracted object graph. Formally, an OGraph is a *sound* approximation of any ROG, represented by a well-typed store S, if the OGraph relates to the ROG as follows:

Object soundness. There is a map H that maps each object ℓ in S to exactly one representative OObject in the OGraph. Similarly, there is a map K such that each runtime domain $\ell.d$ has exactly one representative ODomain in the OGraph.

Edge soundness. If there is a dataflow communication from an object ℓ_1 to ℓ_2 in a ROG, with their representatives OObjects O_1 and O_2 in the OGraph, then there are two maps L_I and L_E that map the pair (ℓ_1, ℓ_2) to a set of OEdges in the OGraph that represent the dataflow communication between O_1 and O_2. Moreover, if the runtime dataflow edge from ℓ_1 to ℓ_2 refers to a runtime object ℓ_3 in a ROG, with its representative OObject O_3 in the OGraph, then there is an OEdge from O_1 to O_2 that refers refers to O_3 (Fig. 2.6, page 21).

To relate the dynamic and the static semantics of the analysis, we define an approximation relation (Df-Approx) between a runtime state (S,H,K,L_I,L_E) and an analysis result (DO, DD, DE). It ensures that the runtime objects, runtime domains and runtime edges are consistent with their representatives in the statically extracted OGraph.

Approximation Relation (Df-Approx).

$$
\begin{aligned}
&\forall\ \Sigma \vdash S,\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE)\\
&\iff\\
&\forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle\\
&\implies\\
&H[\ell] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO\\
&\mathit{and}\ \forall \ell'_j.d_j \in \overline{\ell'.d}\ K[\ell'_j.d_j] = D_j = \langle D_{id_j}, qual(\ell'_j.d_j)\rangle \in rng(DD)\\
&\mathit{and}\ \forall d_i \in domains(C\langle\overline{\ell'.d}\rangle)\\
&\qquad K[\ell.d_i] = D_i = \langle D_{id_i}, C{::}d_i\rangle\ \{(O_C, C{::}d_i) \mapsto D_i\} \in DD\\
&\mathit{and}\ \forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}\\
&\qquad \forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R\\
&\qquad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}\\
&\qquad\qquad H; K; L_I; L_E \vdash O_k \in irLookup(T_k)\\
&\qquad\qquad\quad E'_k \in L_I[(\ell_{src}, \ell)]\ E'_k = \langle H[\ell_{src}], H[\ell], O_k, Imp\rangle \in DE\\
&\mathit{and}\ \forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}\\
&\qquad \forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R\\
&\qquad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}\\
&\qquad\qquad H; K; L_I; L_E \vdash O_k \in irLookup(T_k)\\
&\qquad\qquad\quad E_k \in L_E[(\ell, \ell_{dst})]\ E_k = \langle H[\ell], H[\ell_{dst}], O_k, Exp\rangle \in DE
\end{aligned}
$$

Df-Approx states that given a well-typed store S of a program and an OGraph $G = \langle DO, DD, DE\rangle$ of the same program, there are maps H, K, L_I, and L_E, such that H maps each runtime object ℓ in the store to a unique OObject O_C from DO, K maps each runtime domain $\ell.d_i$ in the store to a unique ODomain D_i, and L_I and L_E map each pair of runtime objects (ℓ_{src}, ℓ) and (ℓ, ℓ_{dst}) to OEdges from DE. Df-Approx ensures the consistency of these mappings with the ownership relation, and with the dataflow communication.

The last two conditions relate runtime dataflow communication back to field reads, field writes, and method invocations that produce the corresponding import and export edges in DE. L_I maps a runtime dataflow communication from a runtime object ℓ_{src} to another runtime object ℓ back to an import $\mathsf{OEdge}\ E'_k$ from DE. By our definition of import dataflow communication, E'_k exists in DE due to a field read or a method invocation expression that has ℓ_{src} as its receiver. The condition also ensures that if a dataflow edge refers to an $\mathsf{OObject}$ of a subtype of T_k, then T_k is the type of a field of ℓ_{src} or the return type of a method of ℓ_{src}.

Similarly, L_E maps a runtime dataflow communication from a runtime object ℓ to another runtime object ℓ_{dst} back to an export $\mathsf{OEdge}\ E_k$ from DE. By our definition of export dataflow communication, E_k exists in DE due to a field write or a method invocation expression that has ℓ_{dst} as its receiver. The condition also ensures that if the dataflow edge refers to an $\mathsf{OObject}$ of a subtype of T_k, then T_k is the type of a field of ℓ_{dst} or the type of a parameter of a method of ℓ_{dst}.

Theorem: Dataflow Object Graph Soundness.

$$
\begin{aligned}
&\textit{If } G = \langle DO, DD, DE \rangle \\
&\quad G \vdash (CT, e_{root}) \\
&\quad \forall e,\ \theta_0 \vdash e; \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E \\
&\quad \Sigma \vdash S \\
&\textit{then } G \vdash_{CT,H} \Sigma \\
&\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE)
\end{aligned}
$$

where $\rightsquigarrow^*_G$ relation is the reflexive and transitive closure of $\rightsquigarrow_G$ relation, and θ_0 is the location of the first object instantiated by e_{root}. To prove the Object Graph Soundness theorem, we prove the Dataflow Preservation and Dataflow Progress theorems, which extend the standard FDJ Preservation and Progress. The common parts are $\boxed{highlighted}$.

Theorem: Dataflow Preservation (Subject reduction).

$$
\begin{aligned}
&\textit{If } \boxed{\emptyset, \Sigma, \theta \vdash e : T} \\
&\quad \boxed{\Sigma \vdash S} \\
&\quad G = \langle DO, DD, DE \rangle \\
&\quad G \vdash_{CT,H} \Sigma \\
&\quad \emptyset, \emptyset, G \vdash_O e \\
&\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
&\quad \theta \vdash \boxed{\text{e; S}}; H; K; L_I; L_E \rightsquigarrow_G \boxed{\text{e'; S'}}; H'; K'; L'_I; L'_E \\
&\textit{then } \boxed{\textit{there exists } \Sigma' \supseteq \Sigma \textit{ and } T' <: T \textit{ such that}} \\
&\quad \boxed{\emptyset, \Sigma', \theta \vdash e' : T' \textit{ and } \Sigma' \vdash S'} \\
&\quad (S', H', K', L'_I, L'_E) \sim (DO, DD, DE) \\
&\quad \emptyset, \emptyset, G \vdash_O e' \\
&\quad \textit{and } G \vdash_{CT,H} \Sigma'
\end{aligned}
$$

Proof. Proof is by induction on the instrumented evaluation relation (Appendix, page 179).

$$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$$

The most interesting cases are Ir-New, Ir-Read (page 182), Ir-Write (page 184), and Ir-Invk (page 186). □

Theorem: Dataflow Progress.

$$
\begin{array}{l}
\textit{If } \boxed{\emptyset, \Sigma, \theta \vdash e : T} \\
\quad \boxed{\Sigma \vdash S} \\
\quad G = \langle DO, DD, DE \rangle \\
\quad G \vdash_{CT,H} \Sigma \\
\quad \emptyset, \emptyset, G \vdash_O e \\
\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
\textit{then either } \boxed{e \textit{ is a value}} \\
\quad \textit{or else } \theta \vdash \boxed{e; S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{e'; S'}; H'; K'; L'_I; L'_E
\end{array}
$$

Proof. Proof is by induction over the derivation of

$$\emptyset, \emptyset, G \vdash_O e$$

with a case analysis on the last typing rule used (Appendix, page 179). The most interesting cases are Df-New, Df-Read (page 198), Df-Write (page 201), and Df-Invk (page 203).

□

3.3 Extensions of Ownership Domains

This section describes more precisely how the analysis handles extension of Ownership Domains [10], which Section 2.7 (page 40) describes informally. In extensions of Ownership Domains, not all the objects are in named, conceptual group. Some objects are borrowed from one domain to another and are aliased by variables declared `lent` in the code. The type system can also guarantee that variables declared `unique` can alias an object that is passed linearly without having a persistence reference to it. The analysis needs to keep track of assignments between variables and uses a value flow graph. During extraction of the OGraph, the analysis attempts to resolve `lent` and `unique` to a domain parameter (α), a locally declared domain ($n.d$), or `shared`. If the attempt fails, the analysis creates a special domain and a flow object in it.

I proved the soundness of the analysis that extracts an OGraph with dataflow edges that refer to dataflow objects (Appendix 8.3, page 179). Flow objects maintain the unique representative invariant since the analysis creates a fresh ODomain for each flow object. Multiple dataflow edges can then refer to the same flow object. I conjecture but do not prove that the extraction analysis with the value flow analysis extracts an OGraph that is also sound. Based on the value flow analysis, ScoriaX adds more objects and more edges. Even if it fails to resolve `lent` and `unique`, ScoriaX creates special domains and flow objects to preserve soundness.

This section describes the revised syntax, data types and static semantics. In the rest of the section, I *highlight* the changes in the formalization that are required to handle the extensions of Ownership Domains.

3.3.1 Extended Syntax

To handle complex expressions in the code, I describe the analysis over the three-address code [128] of an FDJ program (Fig. 3.9). Each three-address code expression has at most

three operands and is typically a combination of an assignment and a binary operator or a method invocation that has only variables as arguments.

The syntax includes `lent` and `unique` [12], and according to FDJ formalization [10], only the first domain parameter (the owner domain) of a type can be `lent` or `unique` because the class `Object` takes one domain parameter, and according to *cdef*, only the first domain is mandatory for every type. An object creation expression can have the first domain parameter `unique` (but not `lent`), thus the syntax uses the meta-variable A for a `new` expression. A type T consists of a class C parameterized with a list of domains that are of the following form: a domain parameter α, a declared domain $n.d$, or `shared`. The syntax also includes the meta-variable T_A for types in which the owner domain can also be `unique`, and T_B for types in which the owner domain can be also `lent`. For example, the type of fields in the class definition cannot be `lent` because a borrowed object cannot be stored in a field; hence, the field type is T_A. On the other hand, the owner domain of the type of a method parameter can be `lent` or `unique`, and the parameter type is T_B.

3.3.2 Extended Object Graph

In addition to the OGraph described in Section 3.1.2, the analysis also builds a value flow graph (FG) that tracks the assignment between two variables x and y, the substitution of the actual argument to the formal method parameter, and object returned from a method. A node in FG is a triplet (O, x, B) that denotes a variable x that is part of an expression that the analysis interprets in the context of O, and x is of a type T_B where the owner domain B is a domain p, `unique`, or `lent`. For brevity, the notation $(O, \overline{x}, \overline{B})$ means a list of n triplets (O, x_1, B_1), ..., (O, x_n, B_n). An edge in FG has a *label* to track if the value flow is due to a method invocation ($\overset{(_i}{\rightsquigarrow}$), a method return ($\overset{)_i}{\rightsquigarrow}$). The label $\bullet$ denotes an empty annotation on a value flow edge due an assignment ($\rightsquigarrow$). The label $\star$ denotes an information flow due to a field write. The label on a value flow edge tracks call-site sensitivity [76, 75]. By considering O as a part of the node, the flow analysis is also domain-sensitive and can

$$
\begin{array}{rcll}
CT & ::= & \overline{cdef} & \textit{table of class declarations} \\
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} & \\
 & & \{\ \overline{dom};\ \ \overline{T_A}\ \overline{f};\ \ C(\overline{T'_A}\ \overline{f'},\overline{T_A}\ \overline{f}) & \\
 & & \{\texttt{super}(f');\texttt{this}.\overline{f}=\overline{f};\}\ \ \overline{md}\ \} & \textit{class decl.} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; & \textit{domain decl.} \\
md & ::= & T_A\ ret\ m(\overline{T_B}\ \overline{x})\ \ T_{this}\ \{ret = e_R;\texttt{return}\ ret;\} & \\
 & & & \textit{method decl.} \\
e & ::= & \mid x = \texttt{new}\ C\texttt{<}A,\overline{p}\texttt{>}(\overline{y}) & \\
 & & \mid x = y.f\ \mid x.f = y \mid x = y\ \mid x = r.m(\overline{y}) & \\
 & & \mid \ell\ \mid \ell \triangleright e & \textit{expressions} \\
n & ::= & x \mid v & \textit{values or variable names} \\
p & ::= & \alpha \mid n.d \mid \texttt{SHARED} & \textit{domain name} \\
A & ::= & \texttt{unique} \mid p & \textit{domain may be unique} \\
B & ::= & \texttt{lent} \mid A & \textit{domain may be lent or unique} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} & \textit{precise type} \\
T_A & ::= & C\texttt{<}A,\overline{p}\texttt{>} & \textit{owner domain may be unique} \\
T_B & ::= & C\texttt{<}B,\overline{p}\texttt{>} & \textit{owner domain may be lent or unique} \\
v,\ell,\theta & \in & \textit{locations} & \\
x,y,r,a & \in & \textit{variables} & \\
S & ::= & \ell \rightarrow C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) & \textit{location store} \\
\Sigma & ::= & \ell \rightarrow T & \textit{store typing} \\
\Gamma & ::= & x \rightarrow T_B & \textit{type environment}
\end{array}
$$

Figure 3.9: Three-address code version of the FDJ syntax extended using `lent` and `unique` [12]. For readability, repeated from Fig. 2.18

$$
\begin{array}{lll}
FG & ::= \emptyset \mid FG \cup \{\ (O_{src}, x, B_{src}) \overset{label}{\rightsquigarrow} (O_{dst}, y, B_{dst})\ \} & \textit{Value Flow Graph} \\
label & ::= (_i \mid)_i \mid \bullet \mid \star & \textit{value flow annotations}
\end{array}
$$

Figure 3.10: Value flow graph data type. For clarity, repeated from Fig. 2.19.

have different nodes for the same variable x if analyzed in different contexts. Section 3.3.4 describes in more detail the role of the *label* in computing the transitive flow through a method invocation or a field assignment.

3.3.3 Extended Static Semantics

The analysis starts as previously described in Section 3.1.3, and also initializes FG with an empty set. The form of the rules is adapted to incorporate FG to the left hand side of the turnstile:

$$\Gamma, \Upsilon, FG, G \vdash_O e$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ O = C_{\texttt{this}}\texttt{<}\overline{D_O}\texttt{>}\qquad \forall i \in 1..|\overline{p}|\qquad FG, G \vdash_O D_i \in findD(C_{\texttt{this}}{::}p_i) \\ O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle\qquad \{O_C\} \subseteq DO \\ dparams(C, O_C)\qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD\qquad \Upsilon, G \vdash_O ddomains(C, O_C) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \{(O, x, p_1) \rightsquigarrow (O_C, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \rightsquigarrow (O_C, \texttt{this}.\overline{f}, owner(\overline{T}))\} \subseteq FG \\ \forall m \in \overline{md}.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, FG, G \vdash_{O_C} e_R\end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{a})}\ [\textsc{Df-New}]$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ O = C_{\texttt{this}}\texttt{<}\overline{D_O}\texttt{>}\qquad FG, G \vdash_O D_1 \in uniqueDomains(C) \\ \forall i \in 2..|\overline{p}|\ FG, G \vdash_O D_i \in findD(C_{\texttt{this}}{::}p_i) \\ O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle\qquad \{O_C\} \subseteq DO \\ dparams(C, O_C)\qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD\qquad \Upsilon, G \vdash_O ddomains(C, O_C) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \{(O, x, p_1) \rightsquigarrow (O_C, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \rightsquigarrow (O_C, \texttt{this}.\overline{f}, owner(\overline{T}))\} \subseteq FG \\ \forall m \in \overline{md}.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, FG, G \vdash_{O_C} e_R\end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\texttt{<}\texttt{unique}, \overline{p}\texttt{>}(\overline{a})}\ [\textsc{Df-New-Unique}]$$

$$\frac{\begin{array}{c}CT(C_r) = \texttt{class}\ C_r\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'_r\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \} \\ G = \langle DO, DD, DE\rangle\qquad \Gamma[r] = T_r = C_r\texttt{<}\overline{p}\texttt{>}\qquad (T_k\ f_k) \in fields(T_r)\qquad FG, G \vdash_O import(T_r, T_k) \\ FG, G \vdash_O O_r \in lookup(T_r)\qquad (T'_k\ f_k) \in \overline{Tf} \\ \{(O, r, owner(T_r)) \rightsquigarrow (O_r, \texttt{this}, \alpha_0), (O_r, f_k, owner(T'_k)) \rightsquigarrow (O, x, owner(T_k))\} \subseteq FG\end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = r.f_k}\ [\textsc{Df-Read}]$$

$$\frac{\begin{array}{c}CT(C_x) = \texttt{class}\ C_x\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'_x\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \} \\ G = \langle DO, DD, DE\rangle\qquad \Gamma[x] = T_x = C_x\texttt{<}\overline{p}\texttt{>}\qquad (T_k\ f_k) \in fields(T_x) \\ \Gamma[r] = T_r\qquad T_r <: T_k\qquad FG, G \vdash_O export(T_x, T_r) \\ FG, G \vdash_O O_x \in lookup(C_x\texttt{<}\overline{p}\texttt{>})\qquad (T'_k\ f_k) \in \overline{Tf} \\ \{(O, r, owner(T_r)) \overset{\star}{\rightsquigarrow} (O_x, f_k, owner(T'_k))\} \subseteq FG\end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x.f_k = r}\ [\textsc{Df-Write}]$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ \Gamma[r_0] = C\texttt{<}\overline{p}\texttt{>}\qquad mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T} \to T_R\qquad FG, G \vdash_O import(C\texttt{<}\overline{p}\texttt{>}, T_R) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \overline{T_a} <: \overline{T}\qquad FG, G \vdash_O export(C\texttt{<}\overline{p}\texttt{>}, \overline{T_a}) \\ FG, G \vdash_O O_r \in lookup(C\texttt{<}\overline{p}\texttt{>})\qquad T'_R\ ret\ m(\overline{T_B}\ \overline{x})\ \ T_{this}\ \{ret = e_R; \texttt{return}\ ret;\} \in \overline{md} \\ i = fresh_i(O, x_0 = r_0.m(\overline{a})) \\ \Gamma[\overline{x}] = \overline{T_x}\qquad \{(O, r_0, p_0) \overset{(_i}{\rightsquigarrow} (O_r, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \overset{(_i}{\rightsquigarrow} (O_r, \overline{x}, owner(\overline{T_x}))\} \subseteq FG \\ \Gamma[ret] = T'_R\qquad \{(O_r, ret, owner(T'_R)) \overset{)_i}{\rightsquigarrow} (O, x_0, owner(T_R))\} \subseteq FG\end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x_0 = r_0.m(\overline{a})}\ [\textsc{Df-Invk}]$$

$$\frac{\Gamma[r] = T_r\qquad \Gamma[x] = T_x\qquad \{(O, r, owner(T_r)) \rightsquigarrow (O, x, owner(T_x))\} \subseteq FG}{\Gamma, \Upsilon, FG, G \vdash_O x = r}\ [\textsc{Df-Assign}]$$

Figure 3.11: Static semantics of the extraction analysis. Highlighted are the parts that construct the value flow graph.

The analysis uses expressions given in the form of three-address code, where x represents the left-hand-side of the expression. In *Df-New*, the analysis interprets an object allocation expression in the context of O. The analysis first ensures that DO contains an OObject O_C for the newly allocated object and DD has the mapping to construct the object-domain hierarchy. The rule *Df-New* also ensures that FG includes an value flow edge from x to `this` and value flow edges from each of the object allocation arguments $\overline{a}$ to the corresponding fields `this`.$\overline{f}$ (Fig. 3.11). Next, *Df-New-Unique* handles the case when the owner domain of the instantiated type is `unique`. The rule is similar to *Df-New*, except that it uses the auxiliary judgment *uniqueDomains* to find the actual owner domain of O_C (Fig. 3.12). If *uniqueDomain* cannot find an actual ODomain, the analysis ensures that an OObject is created in a fresh ODomain, as a child of O. All the child OObjects of such an ODomain are flow objects.

Next, *Df-Read*, *Df-Write*, *Df-Invk*, and *Df-Assign* ensure value flow edges are created in FG. For example, *Df-Read*, ensures that the flow graph contains one edge from the receiver r in the context O to the context variable `this` in the context O_r, and a second edge from the field f_k in the context of O_r to x in the context of O. For a method invocation, the rule *Df-Invk* adds labels to the value flow edges to uniquely identify a method invocation based on the pair $(O, x_0 = r_0.m(\overline{a}))$. For *Df-Read*, *Df-Write*, and *Df-Invk*, the context OObject of the source is different from the context OObject of the destination of a flow edge. On the other hand, for *Df-Assign*, the context OObject remains unchanged for the source and destination.

The rules *Df-Lookup-Lent* and *Df-Lookup-Unique* handle the cases where the owner domain is `lent` or `unique`. These rules use *solveLent* and *solveUnique* to determine the actual domain p' and the context O' where p' is defined. The analysis includes the context O' in the result to be able to determine the actual domain D'_1 corresponding to `lent` or `unique` because the context might be different from the current context O. For example, a class might be instantiated in the context of O where the owner is `unique`. Next, the reference x in

$$\frac{G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\overline{<D>} \quad O_k \in DO \quad O_k = \langle C\overline{<D>}\rangle \quad C <: C' \quad \forall i \in 1..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}::p'_i) \quad D'_i = D_i}{FG, G \vdash_O O_k \in lookup\ (C'\overline{<p'>})}\ [\textsc{Df-Lookup}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\overline{<D>} \quad O_k \in DO \quad O_k = \langle C<D_1, \overline{D}>\rangle \quad C <: C' \\ \forall i \in 2..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}::p'_i) \quad D'_i = D_i \\ FG \vdash (O', x, A) \in solveLent(O, C') \quad O' = C'_{\texttt{this}}\overline{<D'>} \\ A = \texttt{unique} \Longrightarrow FG \vdash (O'', x, \texttt{unique}) \in findSrcUnique(O', C') \wedge D'_1 = DD[(O'', C'::\texttt{unique})] \wedge D'_1 = D_1 \\ A = p' \Longrightarrow FG, G \vdash_{O'} D'_1 \in findD(C'_{\texttt{this}}::p') \wedge D'_1 = D_1 \end{array}}{FG, G \vdash_O O_k \in lookup\ (C'<\texttt{lent}, \overline{p'}>)}\ [\textsc{Df-Lookup-Lent}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\overline{<D>} \quad O_k \in DO \quad O_k = \langle C<D_1, \overline{D}>\rangle \quad C <: C' \\ \forall i \in 2..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}::p'_i) \quad D'_i = D_i \\ FG \vdash (O', y, A) \in solveUnique(O, C') \quad O' = \langle C'_{\texttt{this}}\overline{<D'>}\rangle \\ A = \texttt{unique} \Longrightarrow FG \vdash (O'', x, \texttt{unique}) \in findSrcUnique(O', C') \wedge D'_1 = DD[(O'', C'::\texttt{unique})] \wedge D'_1 = D_1 \\ A = p' \Longrightarrow FG, G \vdash_{O'} D'_1 \in findD(C'_{\texttt{this}}::p') \wedge D'_1 = D_1 \end{array}}{FG, G \vdash_O O_k \in lookup\ (C'<\texttt{unique}, \overline{p'}>)}\ [\textsc{Df-Lookup-Unique}]$$

$$\frac{FG_P = propagateAll(FG) \quad (O, s, \texttt{unique}) \rightsquigarrow (O', y, \texttt{unique}) \in FG_P \quad s : C \quad C <: C' \quad \nexists (O'', x, \texttt{unique}) \rightsquigarrow (O, s, \texttt{unique}) \in FG_P}{FG \vdash (O, s, \texttt{unique}) \in findSrcUnique(O', C')}\ [\textsc{Aux-Find-Unique}]$$

$$\frac{FG_P = propagateAll(FG) \quad (O, x, \texttt{unique}) \rightsquigarrow (O', y, A) \in FG_P \quad y : C \quad C <: C'}{FG \vdash (O', y, A) \in solveUnique(O, C')}\ [\textsc{Aux-Resolve-Unique}]$$

$$\frac{FG_P = propagateAll(FG) \quad (O', x, A) \rightsquigarrow (O, y, \texttt{lent}) \in FG_P \quad x : C \quad C <: C'}{FG \vdash (O', x, A) \in solveLent(O, C)}\ [\textsc{Aux-Resolve-Lent}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad FG \vdash (O', y, A) \in solveUnique(O, C) \\ A = \texttt{unique} \Longrightarrow (D = \langle D_{id}, C::\texttt{unique}\rangle \wedge \{(O, C::\texttt{unique}) \mapsto D\} \subseteq DD) \\ A = p' \Longrightarrow (O' = \langle C'_{\texttt{this}}\overline{<D'>}\rangle \wedge FG, G \vdash_{O'} D \in findD(C'_{\texttt{this}}::p')) \end{array}}{FG, G \vdash_O D \in uniqueDomains(C)}\ [\textsc{Aux-UniqueDom}]$$

Figure 3.12: Rules for resolving `lent`, and `unique`.

$$\frac{O = \langle C\overline{<D_O>}\rangle \quad n : C_n\overline{<p>} \quad FG, G \vdash_O O_i \in lookup\ (C_n\overline{<p>}) \quad D_i = DD[(O_i, C_n::d)]}{FG, G \vdash_O D_i \in findD(C::n.d)}\ [\textsc{Df-FindD-Public}]$$

$$\frac{O = \langle C\overline{<D_O>}\rangle \quad D_i = DD[(O, C::d_i)]}{FG, G \vdash_O D_i \in findD\ (C::\texttt{this}.d_i)}\ [\textsc{Df-FindD-This}]$$

$$\frac{O = \langle C\overline{<D_O>}\rangle \quad D_i = DD[(O, C::\alpha_i)]}{FG, G \vdash_O D_i \in findD\ (C::\alpha_i)}\ [\textsc{Df-FindD}]$$

$$\frac{}{FG, G \vdash_O D_{\texttt{SHARED}} \in findD\ (::\texttt{shared})}\ [\textsc{Df-FindD-Shared}]$$

Figure 3.13: Auxiliary judgments for inference rules in Fig. 3.11.

the context O is assigned to y and in another context O'. The destination y is declared in an actual domain p'. To determine the actual ODomain for the domain parameter p', the

extraction analysis uses the context of y, namely O', not the context of x.

3.3.4 Flow Graph Analysis

Since a value might be passed linearly through several assignments, the rules *solveLent* and *solveUnique* use another flow graph, FG_P, where the transitive flow is propagated, as direct flow edges may not exist in FG. FG_P is computed in two steps. First, an algorithm summarizes FG into a summary graph $FG*$, then another algorithm propagates the transitive flow for nodes in $FG*$ and computes FG_P. The algorithm *summarize* does not consider the order of the assignments and the flow graph analysis is therefore flow-insensitive. By using the labels of the flow edges, the *summarize* matches the parentheses with the same value i and the flow graph analysis is call-site context-sensitive.

Understanding value flow analysis by working examples

In order to better understand the Flow Graph Analysis, I introduce three similar running examples that each creates objects of the same type. The first two examples (Figure 3.15)

function $summarize(FG)$
  $FG* = FG$
  $WL = \{(O_1, x_1, B_1) \overset{a}{\rightsquigarrow} (O_2, x_2, B_2) \in FG \text{ s.t. } a \text{ is } (_i\}$
  while $WL \neq \emptyset$ **do**
    remove $e_1 : (O_1, x_1, B_1) \overset{a_1}{\rightsquigarrow} (O_2, x_2, B_2)$ from WL
    if a_1 is $(_i$ **then**
      for $e_2 : (O_2, x_2, B_2) \overset{a_2}{\rightsquigarrow} (O_3, x_3, B_3) \in FG*$ **do**
        if $e_3 = concat(e_1, e_2) \notin FG*$ **then**
          add e_3 to $FG*$ and WL
    else
      if a_1 is $\bullet$ or $\star$ **then**
        for $e_2' : (O_0, x_0, B_0) \overset{a_2'}{\rightsquigarrow} (O_1, x_1, B_1) \in FG*$ **do**
          if $e_3' = concat(e_2', e_1) \notin FG*$ **then**
            add e_3' to $FG*$ and WL
  return $FG*$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \rightsquigarrow (O_2, z, B_3)) = (O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, z, B_3)$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{)_i}{\rightsquigarrow} (O_1, z, B_3)) = (O_1, x, B_1) \rightsquigarrow (O_1, z, B_3)$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star}{\rightsquigarrow} (O_3, z, B_3)) = (O_1, x, B_1) \overset{\star}{\rightsquigarrow} (O_3, z, B_3)$

Figure 3.14: The algorithm *summarize* inspired from [75, Fig. 4.16]. Other cases of *concat* do not result in additional edges.

show how the Flow Graph Analysis distinguishes between these objects, and the extraction analysis avoids creating false positive dataflow edges. The example in Figure 3.16 highlights one limitation of the extraction analysis if developers overuse `lent` and `unique`. Each example also shows the extracted OGraph that has OObjects such as ⟨A<DATA,DOM1>⟩ and ⟨A<DATA,DOM2>⟩. The flow graph FG (not shown) has nodes such as (⟨A<DATA,DOM1>⟩, f, F) and (⟨A<DATA,DOM2>⟩, f, F) and the following flow edges:

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{a1}, \texttt{DATA}) \overset{(_{11}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{this}, \texttt{owner})$$

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n1}, \texttt{DOM1}) \overset{(_{11}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{num}, \texttt{F})$$

$$(\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{num}, \texttt{F}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{f}, \texttt{F})$$

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{a2}, \texttt{DATA}) \overset{(_{14}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{this}, \texttt{owner})$$

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{DOM2}) \overset{(_{14}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F})$$

$$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$$

$$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F}) \rightsquigarrow (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F})$$

$$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F}) \overset{)_{15}}{\rightsquigarrow} (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$$

Summarizing the value flow graph

The algorithm *summarize* (Fig. 3.14) computes $FG*$ such that it concatenates two edges where the first edge has the same destination as the source of the second edge. The algorithm matches pair of edges $(_i$ and $)_i$ that have the same value for i. If the invocation $(_i$ is followed by an assignment, the algorithm propagates the invocation. The $\star$ label means that a method stores a value in a field. Because other methods can use the value of the field, the $\star$ label cancels the effect of an $(_i$ label and the concatenated edge keeps the $\star$ label. For the example in Fig. 3.15, *summarize* adds the following edges:

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n1}, \texttt{DOM1}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{f}, \texttt{F})$$

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{DOM2}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$$

The flow graph $FG*$ has a transitive flow:

$$(\langle \texttt{A<DATA,DOM2>} \rangle, \texttt{f}, \texttt{F}) \rightsquigarrow \ldots \rightsquigarrow (\langle \texttt{Main<SHARED>} \rangle, \texttt{dest}, \texttt{lent})$$

Therefore, by distinguishing between OObjects of the same type, but with different lists of ODomains $\overline{D}$, the Flow Graph Analysis avoids a false positive.

According to *Aux-Resolve-Lent*, the Flow Graph Analysis resolves `lent` to `F` in the context of $\langle$`A<DATA,DOM2>`$\rangle$. Then, according to *Aux-Lookup-Lent*, *lookup* returns only $\langle$`Integer<DOM2>`$\rangle$ and not $\langle$`Integer<DOM1>`$\rangle$, which would introduce a false positive dataflow edge in the OGraph.

Related work [75, Fig. 4.16] treats fields differently from local variables, and requires a separate may-alias analysis for finding variables that may alias the same receiver object to substitute a field `this.f` to `a1.f` or `a2.f`. In a flow node (O, `this.f`, B), the receiver `this` refers to O, so no separate may-alias analysis is required.

Distinguishing between different method invocations

To compute the index i used on the value flow edges, a naive flow analysis could use the line number of the method invocation expression in the code. If the analysis were to use such a value for i in the rule *Df-Invk*, it would use the same value of i for different values of O, which would create false positive flow edges. Instead, *Df-Invk* uses $fresh_i$ to generate distinct values for i based on the pair (O, $x = r.m(\overline{y})$) and allows the analysis to distinguish between the same method invocation but in different contexts.

For example, consider Fig. 3.15 that shows code fragments where an object of type `B` creates an object of type `A` and invokes the method `set` to assign the value for the field f. The analysis creates two objects `a:A` in different domains for the same object allocation expression `new A()`. The assignment of the field value also occurs at the same method invocation `a.set(n)`. Due to different values returned by $fresh_i$, the analysis considers the method invocation `a.set(n)` twice, first in the context of `b1:B` and second in the context

```
1  class A<owner,F> {
2    Integer<F> f;
3    void set(Integer<F> num) {
4      f = num;
5    }
6    Integer<F> get() { return f; }
7  }
8  class Main<owner>{
9    void main() {
10    domain DATA,DOM1,DOM2;
11    Integer<DOM1> n1= new Integer(-1);
12    A<DATA,DOM1> a1 = new A();
13    a1.set(n1);
14    Integer<DOM2> n2 = new Integer(2);
15    A<DATA,DOM2> a2 = new A();
16    a2.set(n2);
17    Integer<lent> dest = a2.get();
18    dest.compareTo(n1);
19  }
```

```
1  class A<owner,F> {
2    Integer<F> f;
3    void set(Integer<F> num) {
4      f = num;
5    }
6    Integer<F> get() { return f; }
7  }
8  class B<owner, F> {
9    domain OWNED;
10   A<OWNED,F> A a = new A();
11   void assign(Integer<F> n) {
12     a.set(n);
13   }
14 }
15 class Main<owner>{
16   void main() {
17    domain DATA,DOM1,DOM2;
18    Integer<DOM1> n1 = new Integer(-1);
19    B<DATA,DOM1> b1 = new B();
20    b1.assign(n1);
21    Integer<unique> n2 = new Integer(2);
22    B<DATA,DOM2> b2 = new B();
23    b2.assign(n2);
24    Integer<lent> dest = b2.a.get();
25    dest.compareTo(n1);
26   }
27 }
```

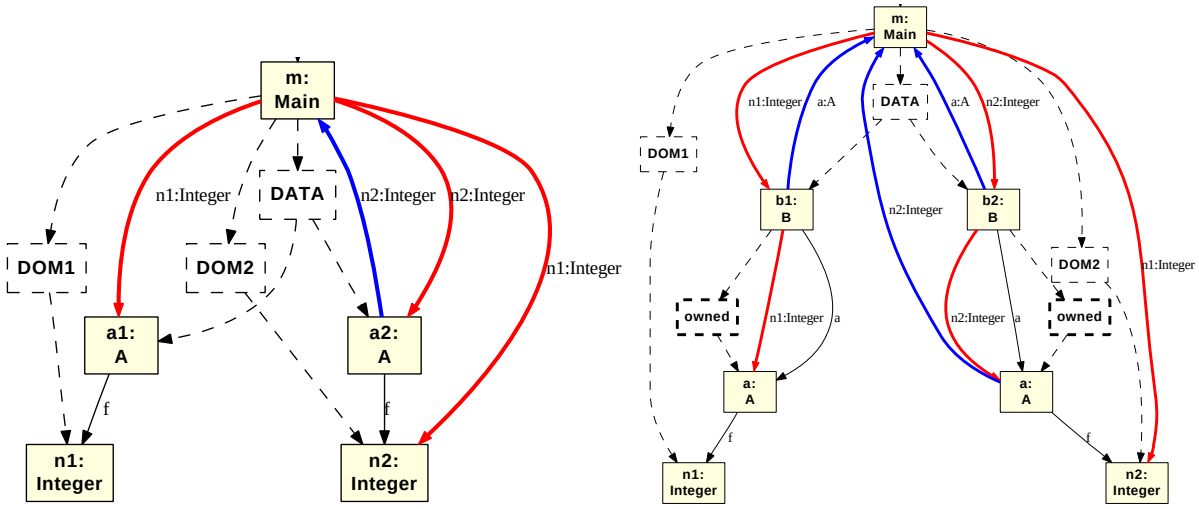


Figure 3.15: The analysis distinguishes between different objects of the same type `A`, and shows a dataflow edge that refers to `n2:Integer` from `a2:A` to `m:Main` and not from `a1:A` to `m:Main`. It is not necessary that the objects of type `A` are created for different object allocation expressions in the code (left vs. right).

```
1  class A<owner,F> {
2    Integer<F> f;
3    void set(Integer<F> num) {
4      f = num;
5    }
6    Integer<F> get() { return f; }
7  }
8  class B<owner, F> {
9    domain OWNED;
10   A<OWNED,F> A a = new A();
11   void assign(Integer<F> n) {
12     a.set(n);
13   }
14 }
```

```
15 class Main<owner>{
16   void main() {
17     domain DATA,DOM1,DOM2;
18     Integer<unique> n1 = new Integer(-1);
19     B<DATA,DOM1> b1 = new B();
20     b1.assign(n1);
21     Integer<unique> n2 = new Integer(2);
22     B<DATA,DOM2> b2 = new B();
23     b2.assign(n2);
24     Integer<lent> dest = b2.a.get();
25     dest.compareTo(n1);
26   }
27 }
```

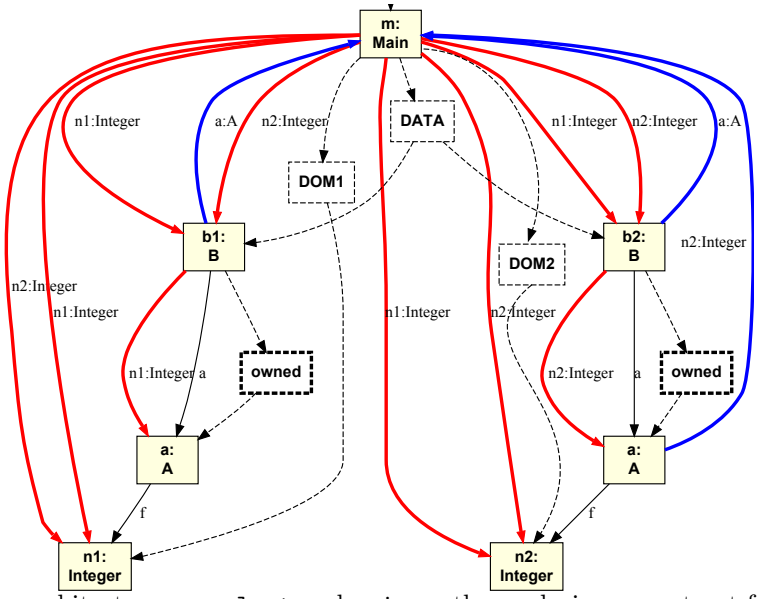


Figure 3.16: If the architects overuse `lent` and `unique`, the analysis may extract false positive edges

of `b2:B`. If the analysis were to consider only the line number of the method invocation as the value of i, $FG*$ would have a false positive transitive flow from (⟨`A<DATA,DOM1>`⟩,`f`,`F`) to (⟨`Main<SHARED>`⟩,`dest`,`lent`).

Computing transitive flow graph

To compute the transitive flow, the analysis uses the algorithm *propagate* (Fig. 3.17), which takes as input the flow graph $FG*$ returned by the *summarize* algorithm and a source s. The output of *propagate* is a transitive flow graph FG_P with the same nodes as FG and $FG*$, but more edges. To compute FG_P, *propagate* uses a worklist algorithm and

the method *concat′*, which concatenates two edges where the destination of the first edge is the source of the first edge. For concatenation, *propagate* uses two value flow labels *Call* and *nCall*. During the initialization, the label *Call* corresponds to flow edges with label $(_i$, and *nCall* correspond to the other label $)_i$. If the second argument of *concat′* is an edge tagged $(_i$, the result is an edge tagged *Call*. Otherwise, either no edge is added, or the concatenation propagates the *nCall* label.

Since the analysis uses the result of the algorithm to resolve `lent` and `unique`, the propagation occurs only if any node of the two edges have a domain B that is `lent`, `unique`, or a public domain $n.d$. The case $n.d$ is needed because a transitive value flow may exist from (O_1, x,$n.d$) to (O_1, y,`lent`), and further to (O_2, z,`lent`), where the variables n and z are in different contexts O_1 and O_2. Here, the analysis performs an extra step and finds the edge from (O,ret,`this`.d) to (O_1, x,$n.d$) such that `this`.d is a domain of O. For the example in Fig. 3.15, where the variable `n2` is declared `unique`, the algorithm *propagate* adds the flow edges:

$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F})$$
$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$$
$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F})$$
$$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$$

Then, *Df-New-Unique* invokes *findD*(`Main::unique`) that in turn invokes *solveUnique*(⟨`Main<SHARED>`⟩,`Integer`). Here, `unique` is resolved to the domain parameter `F` that is bound to `DOM2` in the context of ⟨`B<DATA,DOM2>`⟩. Therefore, *Df-New-Unique* creates the OObject ⟨`Integer<DOM2>`⟩.

3.4 Implementation

The extraction analysis follows the algorithm *runAnalysis* (Fig. 3.18) and starts from the root expression that is provided as input along with all the class declarations in the program. For initialization, *runAnalysis* uses the initial values FG_0, DD_0, DD_0, and DE_0.

function *propagate*($FG*$, s)
  $FG_P = \emptyset$
  $WL = \emptyset$
  for $e : (O_1, s, B_1) \overset{a}{\rightsquigarrow} (O_2, x_2, B_2) \in FG*$ **do**
    if $\{B1, B2\} \cap \{\texttt{lent}, \texttt{unique}, n.d\} \neq \emptyset$ **then**
      if a_1 is $(_i$ **then**
        add $(O_1, s, B_1) \overset{Call}{\rightsquigarrow} (O_2, x_2, B_2)$ to FG_P and WL
      else
        add $(O_1, s, B_1) \overset{nCall}{\rightsquigarrow} (O_2, x_2, B_2)$ to FG_P and WL
  while $WL \neq \emptyset$ **do**
    remove $e_1 : (O_1, x_1, B_1) \overset{nCall|Call}{\rightsquigarrow} (O_2, x_2, B_2)$ from WL
    for $e_2 : (O_2, x_2, B_2) \overset{a}{\rightsquigarrow} (O_3, x_3, B_3) \in FG*$ **do**
      if $e_3 = concat'(e_1, e_2) \notin FG_P$ **then**
        add e_3 to $FG*$ and WL
  return FG_P

function *propagateAll*(FG)
  $FG* = summarize(FG)$;
  for $(O, x, B) \in FG*$ **do**
    $FG_P = FG_P \cup propagate(FG*, x)$
  return FG_P

$concat'((O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{(_i}{\rightsquigarrow} (O_2, z, B_3)) = (O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, z, B_3)$

$concat'((O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star|\bullet|)_i}{\rightsquigarrow} (O_1, z, B_3))$ NO Edge

$concat'((O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{(_i}{\rightsquigarrow} (O_3, z, B_3)) = (O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_3, z, B_3)$

$concat'((O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star|\bullet|)_i}{\rightsquigarrow} (O_3, z, B_3)) = (O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_3, z, B_3)$

Figure 3.17: The algorithm *propagate* adds more edges between nodes for variables declared as `lent` or `unique`.

$$D_{\texttt{SHARED}} = \langle D_0, ::\texttt{SHARED} \rangle, O_{world} = \langle C_{dummy} < . > \rangle$$

$$FG_0 = \emptyset,\ DO_0 = \{O_{world}\}$$

$$DD_0 = \{(O_{world},\ ::\texttt{shared}) \mapsto D_{\texttt{shared}}\}, DE_0 = \emptyset$$

The algorithm has two iterations. First, the analysis creates the object hierarchy with OObjects and ODomains only, and collects the nodes and edges of the value flow graph FG. The iteration terminates when it reaches a fixed point, i.e., no flow nodes and flow edges are added to FG, and no new objects and domains are added to the OGraph. In this iteration, the extraction analysis does not compute dataflow edges and flow objects, because computing dataflow edges requires FG, which further depends on DO. Otherwise, the analysis may

extract a spurious number of dataflow edges that refer to these flow objects when it is not able to resolve `lent` and `unique` based on the intermediate FG. In the second iteration, the analysis invokes the algorithm *propagateAll*, creates the flow objects, and computes the dataflow edges. The second iteration terminates when the algorithm *propagateAll* no longer adds edges to FG_P and no objects, domains, or dataflow edges are added to the OGraph.

Section 3.3 describes the analysis using a constraint-based specification which are declarative, while the implementation uses transfer functions which are imperative. Each function *accept* is a transfer function that is equivalent to the corresponding inference rule. It takes as arguments all the elements on the left-hand-side of the turnstile in the constraint-based specification and the context OObject O. For each declarative clause, $subset \subseteq set$ in the constraint-based specification, the transfer function has an statement $set' = subset \cup set$. In particular, for each object allocation expression, the function *accept* creates a new OObject O_C. Hence, the clause that ensures $\{O_C\} \subseteq DO$ in *Df-New* becomes the statement $DO' = \{O_C\} \cup DO$ in the transfer function. Next, *accept* abstractly interprets each method declaration of the instantiated class, and of the class it extends, recursively. The algorithmic description omits the remaining details of *Df-New* (Fig. 3.11). The transfer functions for the other expressions are similar.

function *runAnalysis*(e_{root}, CT)
$DO = DO_0$, $DD = DD_0$, $DE = DE_0$ $FG = FG_0$, $\Gamma = \emptyset$, $\Upsilon = \emptyset$
$G = \langle DO, DD, DE \rangle$ $G' = \langle DO', DD', DE' \rangle$
$G' \subseteq G \Leftrightarrow DO' \subseteq DO \wedge DD' \subseteq DD \wedge DE' \subseteq DE$
// *Iteration 1*
$\langle FG', G' \rangle = accept(\langle \Gamma, \Upsilon, FG, G, O_{world} \rangle, e_{root})$
while $FG' \subseteq FG \wedge G' \subseteq G$ **do**
$\langle FG, G \rangle = \langle FG', G' \rangle$
$\langle FG', G' \rangle = accept(\langle \Gamma, \Upsilon, FG, G, O_{world} \rangle, e_{root})$
// *summarize FG*
$FG_P = propagateAll(FG)$
// *Iteration 2*
$\langle FG', G' \rangle = accept(\langle \Gamma, \Upsilon, FG_P, G, O_{world} \rangle, e_{root})$
$FG_P = propagateAll(FG')$
while $FG' \subseteq FG_P \wedge G' \subseteq G$ **do**
$G = G'$
$\langle FG', G' \rangle = accept(\langle \Gamma, \Upsilon, FG_P, G, O_{world} \rangle, e_{root})$
$FG_P = propagateAll(FG')$
return G

function $accept(\langle \Gamma, \Upsilon, FG, G, O \rangle, x = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}())$
$\langle DO, DD, DE \rangle = G$
$O_C = \langle C\texttt{<}\overline{D}\texttt{>} \rangle, DO' = DO \cup \{O_C\} \ldots$ //as in *Df-New*
for $m \in \overline{md}$. $mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$ **do**
if $C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon$ **then**
$\Gamma' = \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}$
$\Upsilon' = \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}$
$G' = \langle DO', DD', DE' \rangle$
$\langle FG', G' \rangle = accept(\Gamma', \Upsilon', FG', G', O_C, e_R)$
return $\langle FG', G' \rangle$
function $accept(\langle \Gamma, \Upsilon, FG, G, O \rangle, x = \ldots)$

Figure 3.18: The algorithm *runAnalysis* describes the steps the analysis follows while iterating multiple times over the code. An informal version is in Fig. 2.20 (Section 2.7.4, page 45)

3.4.1 Implementation Supports Full Java Language

ScoriaX is implemented as a plug-in for Eclipse developed using the Crystal static analysis framework [104], which allows the development of a static analysis to follow closely the formal description using transfer functions. There are several differences between the formalization of the analysis and the concrete implementation of the analysis. The implementation supports the following features of the Java language not included in FDJ: generic types, interfaces, arrays, nested classes, and cast expressions. The implementation does not support other features of Java such as static code, anonymous class[3], Java reflection and native code.

For example, to support interfaces, since a class can implement multiple interfaces, ScoriaX unifies domain declarations in multiple interfaces based on their names. That is, for a class C that implements interfaces I_1, I_2, I_3 that declares a domain D, $I_1::D = I_2::D = I_3::D$. The analysis treats generics and ownership domains as orthogonal [27] at the cost of more verbose annotations. Figure 2.16 (page 38) gives an example of using Ownership Domains and generics in the class `HashMap`.

3.4.2 Extraction of Other Types of Edges

ScoriaX can be extended to extract other types of edges such as points-to edges, previously formalized in [4, 5], creation edges, and control flow edges. The extraction of object-domain hierarchy remains unchanged, while the data types (Fig. 3.19), and the rules *Df-New* and *Df-Invk* are extended.

ScoriaX handles differently the invocation of methods from the invocation of constructors. When a source object creates a destination object and passes references to the destination as constructor arguments, the analysis extracts a creation edge for each argument. Similarly to a dataflow edge, a creation edge refers to an object. Extracting creation edges require

[3]For anonymous classes, a workaround is to refactor the code using refactoring tool support provided by Eclipse.

$$
\begin{array}{lll}
E \in \mathsf{OEdge} & ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Label} = O_{label}, \mathbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle & \textit{Dataflow Edge} \\
CrE \in \mathsf{OCREdge} & ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Label} = O_{label}\ \rangle & \textit{Creation Edge} \\
PtE \in \mathsf{OPTEdge} & ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Field} = f\ \rangle & \textit{Points-to Edge} \\
CfE \in \mathsf{OCFEdge} & ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Method} = m\ \rangle & \textit{Control Flow Edge} \\
DE & ::= \emptyset\ \mid\ DE \cup \{\ E, CrE, PtE, CfE\ \} & \textit{All Edges}
\end{array}
$$

Figure 3.19: Data types to include additional types of edges: points-to, creation, and control flow edges

$$
\frac{\begin{array}{c} G = \langle DO, DD, DE \rangle \\ O = C_{\texttt{this}}\overline{<D_O>} \qquad \forall i \in 1..|\overline{p}| \qquad FG, G \vdash_O D_i \in findD(C_{\texttt{this}}::p_i) \\ O_C = \langle\ C\overline{<D>}\ \rangle \qquad \{O_C\} \subseteq DO \\ \forall T_a \in \Gamma[\overline{a}] \qquad FG, G \vdash_O O_a \in lookup(T_a) \qquad \{\langle O, O_C, O_a \rangle\} \subseteq DE \\ \cdots \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\overline{<p>}(\overline{a})}\ [\textsc{Df-New}]
$$

$$
\frac{\begin{array}{c} G = \langle DO, DD, DE \rangle \\ O = C_{\texttt{this}}\overline{<D_O>} \qquad FG, G \vdash_O D_1 \in uniqueDomains(C) \\ \forall i \in 2..|\overline{p}|\ \ FG, G \vdash_O D_i \in findD(C_{\texttt{this}}::p_i) \\ O_C = \langle\ C\overline{<D>}\ \rangle \qquad \{O_C\} \subseteq DO \\ \forall T_a \in \Gamma[\overline{a}] \qquad FG, G \vdash_O O_a \in lookup(T_a) \qquad \{\langle O, O_C, O_a \rangle\} \subseteq DE \\ \cdots \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\overline{<\texttt{unique}, p>}(\overline{a})}\ [\textsc{Df-New-Unique}]
$$

$$
\frac{\begin{array}{c} G = \langle DO, DD, DE \rangle \\ \Gamma[r_0] = C\overline{<p>} \qquad mtype(m, C\overline{<p>}) = \overline{T} \to T_R \qquad FG, G \vdash_O import(C\overline{<p>}, T_R) \\ \Gamma[\overline{a}] = \overline{T_a} \qquad \overline{T_a} <: \overline{T} \qquad FG, G \vdash_O export(C\overline{<p>}, \overline{T_a}) \\ FG, G \vdash_O O_r \in lookup(C\overline{<p>}) \qquad \{\langle O, O_r, m \rangle\} \subseteq DE \\ \cdots \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x_0 = r_0.m(\overline{a})}\ [\textsc{Df-Invk}]
$$

Figure 3.20: Static semantics for extraction of creation and control flow edges. For brevity, the rules omit some of the details in Fig. 3.11.

straightforward changes in *Df-New*, where *lookup* is invoked for the type of each argument (Fig. 3.20).

An object may control the state of another object without sending any data to it. For example, an object can start a thread by invoking the method `Thread.start()`. If a method invocation or a constructor has no arguments, the analysis does not create export dataflow edges. Instead, the analysis can create a control flow edge that refers to the invoked method.

3.5 Previous Versions of ScoriaX

An early version of ScoriaX extracted dataflow edges that refer to types and was described using transfer functions. The data type of the OEdge had C instead of the OObject O_{label}.

$$E \in \mathsf{OEdge} \qquad ::= \langle\ \mathbf{From} = O_{src},\ \mathbf{To} = O_{dst}, \mathbf{Label} = C, \mathbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle$$

The OGraph did not support different edges that refer to different objects of the same type. In addition to the straightforward changes in the data types, *Df-Read*, *Df-Write* and *Df-Invk*, having dataflow edges that refer to objects involved changing the definition of soundness to account for dataflow objects, changing the approximation relation by introducing the auxiliary judgment $ir Lookup$, which further propagate the change to the rule *Ir-New*. Consequently, the Progress and Preservation proofs for the cases *Df-New*, *Df-Read*, *Df-Write*, and *Df-Invk* have been updated.

Another major difference with the earlier system is adding support for the extensions of Ownership Domains. The annotations `lent` and `unique` are used in practice to annotate legacy code. Ignoring these annotation results into an unsound OGraph that may be missing objects or edges.

Other changes involve fixing how the analysis handles recursive types (Section 2.8.1). Previous version supports only recursive types that declare one domain, otherwise the analysis does not terminate. To ensure termination the rule *Aux-DOM* checks all objects in the analysis stack Υ that have the same type with the current object being created. The change propagated in the proof of the Progress theorem to the case *Df-New* and further to the lemma Df-Domains (page 209).

Differences with previous work. ScoriaX is similar to the one that extracts an OGraph with points-to edges [5]. The two analyses create the same ownership hierarchy, but different

types of edges. A points-to edge [5] has a label that is just the field name. The key differences deal with generating dataflow edges that refer to objects and the soundness proof [4, Sec. 3.2, and 3.3].

Differences with earlier versions of this work. This thesis revises the rules in the static and dynamic semantics of Rawshdehs thesis [106, 107], and fully defines the approximation relation and proves progress, preservation, and soundness.

3.6 Summary

This chapter formalizes an analysis that extracts dataflow edges that refer to objects in a sound, hierarchical object graph, and proves object soundness and edge soundness such that ScoriaX meets the extraction requirement *H1c*. The analysis is domain-sensitive and meets the aliasing requirement (*H1d*). The analysis supports legacy code (*H1b*) annotated using the extensions of Ownership Domains. The analysis extracts an object-domain hierarchy (*H1a*) and by design ensures a finite depth in the presence of recursive types (*H1f*). To achieve precision (*H1e*), the analysis relies on Ownership Domains such that it distinguishes between objects of the same type in different domains, and between dataflow edges that refer to objects of the same type. The next chapter evaluates the scalability of the analysis on real-world applications and discusses the precision of the extracted graph.

Acknowledgements

I thank Suhib Rawshdeh for his contributions to the earliest version of the formalization to extract dataflow edges, which did not consider the value flow [106, 107].

Chapter 4 Evaluation of Extraction

Previous chapters introduced ScoriaX, a static analysis that extracts dataflow communication edges in a sound, hierarchical object graph. This chapter discusses the evaluation of ScoriaX based on three systems (Table 4.2), provides support for the hypothesis H1 and discusses how ScoriaX meets the extraction requirements (Section 1.5.1, page 10), in particular how ScoriaX meets the precision (*H1e*) and scalability (*H1h*) in practice.

> **H1.** *For an object-oriented program, the architect adds annotations and uses a static analysis to extract dataflow communication edges that refer to objects in a sound, hierarchical object graph that approximates the runtime architecture of the program, and meets the extraction requirements.*

In the rest of the chapter, Section 4.1 describes the tool support used during the evaluation. Section 4.2 describes the methodology. Section 4.3 introduces the subject systems. Section 4.4 describes the results, and Section 4.5 concludes.

4.1 Tool Support

ScoriaX supports legacy code (H1b) and as discussed in Section 3.4 (page 86) ScoriaX is plug-in for Eclipse. As input, the extractor provides the code with annotations, and the root class. ScoriaX can persist an extracted OGraph into an XML file that can be visualized using a standalone viewer [4, Section 4.3.2], which allows expending and collapsing graphs. ScoriaX also supports the `dot` format of the GraphViz tool [50], which supports cluster graphs used to represent the graph using nested boxes but with limited support for user interaction. Most figures of OGraphs in this thesis are generated from exported dot files.

ScoriaX warns the user if the result of the extraction is unsound (H1c) due to the code using some features in Java that the analysis does not currently handle such as static initialization blocks. The extractor can click on an *extraction warning* and go to the line of code

for which the extraction analysis fails. Extraction warnings are also reported if annotations are missing or are inconsistent.

A typechecker ensures that annotations and code are consistent. To help the extractor while adding annotations, ScoriaX uses an existing typechecker [4, Appendix A] and tool support for adding boilerplate annotations [6]. The annotations use language support for annotations available since Java 1.5 and implement the Ownership Domains type system [10]. The typechecker ensures that annotations are consistent with each other and with the code, and reports any inconsistencies as *typechecker warnings* that the annotator needs to address before running ScoriaX. The typechecker is also integrated in Eclipse and reports the warnings in the default Problems Window. Additional details on the annotation language and the typechecker are in [4, Appendix A].

Extraction in the presence of external libraries. ScoriaX is a whole program analysis but the code of libraries and frameworks that the program use may not always be available to be annotated. Even when available, a library or a framework is often larger than the application itself, which may use only a small part of it. The typechecker allows associating Ownership Domain annotations with Java bytecode using external files, which can be reused across systems. The annotator can provide annotations for each class and method declaration in the library that are referred from the annotated code. Only the class declaration, field declarations, method parameters and return values require annotations, while the bodies of the methods are discarded.

4.2 Extraction Methodology

Extracting an object graph is not a push-button operation, and the extractor follows the process in Table 4.1. First, the extractor gathers the documentation about the system to be analyzed, prepares the code and the root class, which instantiates the classes with the entry points in the applications. In this evaluation, the extractor selected the root class from

```
1  @Domains({"UI","LOGIC","DATA"})
2  class Main{
3    public static void main(@Domain("lent[shared]") String[] args) {
4      @Domain("shared") Main m = new Main();
5    }
6    void run() {
7      @Domain("UI<UI,LOGIC,DATA>")
8      MainActivity act = new MainActivity();
9      @Domain("LOGIC") Bundle b = new Bundle();
10      act.onCreate(b);
11   }
12 }
13 @DomainParams({"U","L","D"})
14 @DomainInherits({"Activity<U,L,D>"})
15 class MainActivity extend Activity{ . . . }
```

Figure 4.1: An example of a root class for an Android application.

the existing classes, or prepared it manually based on existing documentation [6]. In future work, it is possible to generate the root class from configuration files as is done by other tools [48]. That is, the root class instantiates every class that is being loaded by reflection from a configuration file.

Figure 4.1 shows an example of a root class for an Android application that usually resides on one machine. To express that an Android app follows the State-Logic-Display architecture, which is a three-tier architecture [122], the names the top-level domains are `UI` (for Display), `LOGIC` (for Logic), and `DATA` (for State), with the corresponding domain parameters `U`, `L`, and `D`. From the existing documentation or configuration file, the annotator can find the first object that the Android framework instantiates (e.g., `MainActivity`).

Since there are multiple ways to annotate a system, the process of adding annotations and extracting the graph is iterative. The annotator can fine-tune the annotations to gain more precision, push objects underneath other objects in the object-domain hierarchy, or pull objects at higher level of the hierarchy. The goal of the extractor is to minimize both typechecker and extraction warnings. It is often useful to prioritize typechecker warnings where missing annotations has a higher priority than adding annotations for library classes [4, Section 4.4.1].

Table 4.1: The process of extracting object graphs and existing tool support

Step	Stage	Task	Tool support
1	Prepare code	Use generics, extract local variables for object allocation expressions that are part of complex expressions, convert anonymous classes into nested classes	Eclipse refactoring tools
2		Create a root class that instantiates the entry points of the program	N/A
3	Add annotations	For every class declaration and every variable declaration in the code, add Ownership Domains annotations	tool support for adding default annotations [6]
4		Minimize annotation warnings	Typechecker [4]
5	Extract object graph	Prepare ScoriaX configuration file, specify root class, entry point, and optionally what types of edges to extract and output format	ScoriaX
6		Minimize extraction warnings	ScoriaX
7		Refine annotations as needed	Typechecker
8		Add summary for library classes	Tool support for associating ownership domain annotations with Java bytecode
9		Repeat from step 6	

4.3 Subject systems

In this thesis, I evaluated ScoriaX on the three systems in Table 4.2, which lists their size. For each system, the implementation of the analysis extracts dataflow edges and flow objects, creation edges, and points-to edges.

Apache FTP Server (AFS) [1] is an open-source implementation of a complete and portable server engine based on the standard File Transfer Protocol (FTP). The annotators were guided by the reference architecture available [129].

CryptoDB (CDB) [69] is a secure database system designed by a security expert that provides cryptographic protections against unauthorized access, and includes a sample implementation in Java. CryptoDB has both a Java implementation and an informal architectural

Table 4.2: Subject systems: Apache FtpServer (AFS), CryptoDB (CDB), and Universal Password Manager for Android (UPMA)

System	Kind	Version	KLOC	Classes	Interfaces	DIT	
						max	mean
AFS	Java app	1.0.5	14.4	159	39	5	1.68
CDB	Java app		2.3	21	1	3	1.57
UPMA	Android app	v1.14	4.2	88	4	7	2.13
Total			**20.9**				

description that guided the process of adding annotations and extracting object graphs [4, Sec. 7.8]. The object graph was then compared against a detailed runtime architecture drawn by a security expert [7, 131].

Universal Password Manager for Android (UPMA)[1] is an Android application that stores the passwords of the users using an encrypted database. The database is stored in a file that can be shared between multiple devices using a standalone server or online storage services such as DropBox. The annotations follow the State-Logic-Display architecture.

4.4 Results

Table 4.3 shows the number of abstract objects, flow objects, and different types of edges that ScoriaX extracts. In each system, about half of the dataflow edges are self-edges. I focus on non-self dataflow edges that are more interesting for reasoning about security. For each system, the OGraphs are sparse, but the dataflow edges show communication that is relevant for security. Multiple abstract objects flow from the same source to the same destination, which enables reasoning about communication size, which measures the number of distinct dataflow edges from a source to a destination. For example, the CDB OGraph shows 4 dataflow edges from the `mgr:ConsumerManager` to the object `provider:Provider`, which handles encryption keys. The edges refer to objects representing customer information and credit card information that is confidential. These edges are true positives and the architects need to ensure that the `provider:Provider` does not disclose credit card information in clear

[1] I analyzed UPMA to check if it satisfies the Android security policies. This chapter provides a quantitative analysis of the extraction, while a more detail evaluation that finds security vulnerabilities is in Section 6.2.

Table 4.3: Size of the extracted OGraph with dataflow edges, points-to edges, and creation edges. Shows also self-edges, multiple dataflow edges between same objects (comm. size), and dataflow edges that refer to the same object (same flow).

System	objects		Dataflow		Creation		Points-to		Communication size			Same flow		
	edges	flow	edges	self	edges	self	edges	self	min	max	mean	min	max	mean
AFS	129	18	642	314	92	3	400	8	0	42	0.0223	0	116	2.984
CDB	49	10	180	86	47	3	104	0	0	4	0.0400	0	49	1.918
UPMA	86	7	232	127	70	0	258	0	0	3	0.0144	0	31	1.221

text further into an untrusted sink.

In addition, multiple dataflow edges refer to the same abstract object (same flow), which enable the architect to track flow of data in the abstract object graph. The CDB OGraph has 5 dataflow edges that refer to encrypted data and 3 dataflow edges that refer to the same object that represents the credit card information. The credit card information flows from an object of type `CustomerManager` to an object of type `CustomerInfo` and back, and then from `mgr:CustomermManager` to `provider:Provider`. The architect needs to ensure that `provider:Provider` does not disclose credit card information to other objects and then investigate all the edges from `cci:CreditCardInfo` to `provider:Provider`. Indeed, one such edge exists, where the provider receives an object of type `HashMap<String,String>` that contains the credit card number and expiration date and are sent to the encryption engine. It is possible then to write a constraint to ensure that no other dataflow edge refer to this object as I will discuss in Chapter 6 (page 117).

4.4.1 ScoriaX Meets the Extraction Requirements

To provide evidences that support the hypothesis, the qualitative analysis focuses on showing how ScoriaX meets the extraction requirements.

H1a: Express design intent

The architect can refine the annotations to express design intent, as needed. In AFS, the architects can check that an anonymous user cannot execute administrator commands, where each command has a corresponding class that implements the interface `Command`. This

may lead a vulnerability because executing a command, such as `RMDIR`, would enable an attacker to remove folders on the server. After adding the initial annotations and examining the extracted OGraph, the architect decides to refine the annotations by making all the administrator commands from the domain `LOGIC` into a separate domain `COMMANDS`. The AFS OGraph has 4 different dataflow edges that refer to an object representing an anonymous user. The architect can then impose the constraint that no dataflow edges that refer to the anonymous user has as a destination an object in the domain `COMMANDS`.

H1c: Sound object graph

The architect assumes that the object graph shows all possible objects and edges that may occur in any possible program run. Fewer than 20% of the objects are flow objects, but resolving `lent` and `unique` is crucial for meeting the soundness requirement (*H1c*). Otherwise, in the presence of flow objects, some runtime objects and edges would have no representative in the extracted object graph. To resolve `lent` and `unique`, the analysis also meets the aliasing requirement (*H1d*) using a separate, flow-insensitive but domain-sensitive analysis. The value flow analysis is crucial for extracting dataflow edges because about 20–40% of variables are declared `lent` and `unique` [134].

H1e: Precise object graph

ScoriaX meets the precision requirement: the extracted graph does not suffer from excessive merging and is not a fully connected graph. In fact, the extracted object graphs are sparse and have a comparable number of objects and non-self dataflow edges. For AFS, the number of non-self dataflow edges is 328, which is one order of magnitude lower than the number of edges in a fully connected graph. Still imprecision occurs if the architects overuse the annotations `shared`, `lent`, and `unique` (Section 3.3.4, page 81).

The AFS OGraph has a maximum of 42 dataflow edges from the protocol interpreter to the client session. These dataflow edges refer to objects that represent user information, files, and

an object of type `ServerSocket`. Since some of these objects represent confidential data, the architect need to ensure that the object of type `ServerSocket` provides encryption to avoid eavesdropping. Tracing to code from the dataflow edge that refers to a `ss:ServerSocket` leads to the method invocation `ioSession.getAttribute(name)`, but a closer inspection reveals that the edge is a false positive. Since the return type of the method `getAttribute` of the class `FtpIoSession` is `Object` and is annotated `shared`, the analysis finds all the objects in `shared` including the object of type `ServerSocket`. To avoid the false positives that may further lead to false positives when executing constraints, the architects can fine-tune the annotations to improve precision. That is, one can change the annotation such that the object of type `ServerSocket` is in `OWNED` instead of `shared`. In general, using `shared` is discouraged since little reasoning can be done about such objects.

H1f: Summarization

By referring to a pair (type, domain) corresponding to an abstract object, the architects can reason about objects that have the same conceptual purpose or the same security property, even if they are created in different locations in the code. In each system, the maximum of dataflow edges that refer to the same object occurs for dataflow edges that refer to the object of type `String` in the domain `shared`, but there are examples that are more interesting. Architects can fine tune annotations and use hierarchy to distinguish between objects of type `String` that have different conceptual purposes. The UPMA OGraph has different objects of type `String` to distinguish between the password and the account name where only the password represents confidential information. There are 4 dataflow edges that refer to the password, and 10 that refer to the account name. The architects can reason about the destination of the dataflow edges that refer to a password, check if it is an untrusted sink, or if another edge with an untrusted sink as a destination refers to the destination.

H1g: High-level view

ScoriaX meets the high-level view requirement (*H1g*) where the object-domain hierarchy helps improving precision of the analysis. The depth of the object-domain hierarchy seems constant compared to the size of the graph. Empirical data [134] shows that object-domain hierarchy achieves 1x-10x reduction in the number of objects in the domains declared by the root class. These objects are architecturally relevant and the object graph fits on one page by showing only these objects with their substructure collapsed and dataflow edges lifted to their first visible ancestor of the source and destination. The architects can still recover the details by expanding the substructure. Reasoning about security requires more than tracking objects visually, and architects need support to query the object graph, which Scoria also provides (Section 5.3, page 110).

H1h: Scalability

The extraction analysis needs to scale to real-world applications of at least a few thousands lines of code. Tradeoffs needs to be considered: the more precise the extraction analysis is, the less scalable it is. ScoriaX achieves scalability (*H1h*) for systems the size of a typical Android app (2–30KLOC) [88], but their size is too small to claim that ScoriaX meets the scalability requirement. Computing transitive value flow is probably a culprit in limiting the scalability of the extraction analysis. Another bottleneck that limits the scalability of the analysis is the effort of adding annotations. For UPMA, I estimate the effort of adding annotations and extracting the object graph to be 2 hours/KLOC, where part of the effort was required to summarize the Android framework (Table 4.4). Conceivably, less effort would be needed to annotate another Android application since some of the summaries can be reused.

Table 4.4: UPMA: Estimated effort to annotate and reason about security.

Phase	**Effort**	**Percent**
Adding annotations	8.5 hours	68%
Extracting object graph	2 minutes	<1%
Writing constraints	1 hour	8%
Refining annotations	3 hours	24%
Total	12.5 hours	100%

4.5 Summary

This chapter shows an evaluation by running ScoriaX on three applications with more than 20KLOC of Java code. The chapter discusses examples of false positive edges due to excessive use of annotations `shared`, but also shows how the architects can increase precision by placing objects of the same type in different domains. Architects can find cases where more precision is needed, and fine-tune the annotations to help the analysis achieve more precision as needed. This chapter shows that ScoriaX can be used for positive assurance and does not report any vulnerability. A more systematic evaluation, where ScoriaX is incorporated in Architectural Risk Analysis, and architects write machine-checkable constraints is in Chapter 6 (page 117).

Chapter 5 The Scoria Approach

This chapter[1] introduces a semi-automated approach named Scoria that finds security vulnerability using constraints on the abstract object graph. The chapter discusses how Scoria meets the security analysis requirements (Section 1.4, page 9) and presents the security model that allows architects to query the object graph and write constraints. By separating constraints from extraction, the model is extensible (**SC3**) such that architect can write application-specific constraints. Since the model extends an object graph that is hierarchical and has dataflow edges that refer to objects, Scoria allows architects to write constraint in terms of object provenance (*H2a*) and indirect communication (*H2f*).

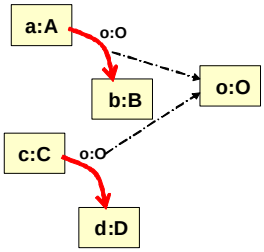
In the following, Section 5.1 gives an overview of Scoria. Section 5.2 defines the concepts used by the approach. Section 5.3 describes a security model, and Section 5.4 concludes.

5.1 Scoria Overview

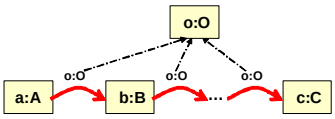
Scoria is a semi-automated approach that statically approximates the runtime architecture of an object-oriented system as a sound abstract object graph to support architects on finding security-relevant architectural flaws. Having a sound abstract graph enables reasoning about abstract object identity, which means that every abstract object of the abstract graph is uniquely identified. Object identity enables the comparison of references. In particular, distinct edges can refer to the same abstract object.

Because the abstract graph tracks when different edges refer to the same abstract object, it enables reasoning about object provenance when the same abstract object that flows from a first source to a first destination also flows from a second source to a second destination (Fig. 5.1(a)). Object transitivity tracks when the same abstract object flows from an initial

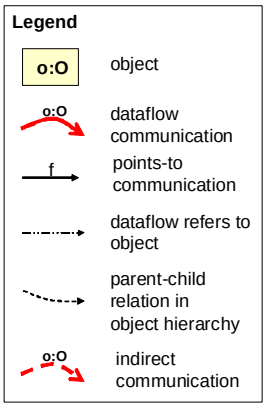
[1] Portions of this chapter appeared in [133].

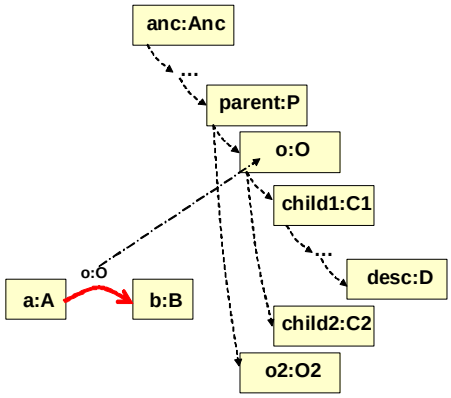


(a) Object Provenance: The same `o:O` that flows from `a:A` to `b:B` should not flow from `c:C` to `d:D`

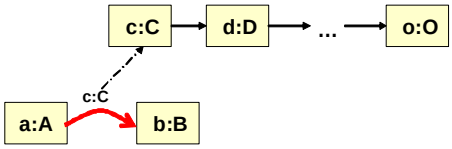


(b) Object Transitivity: All dataflow edges in a path refer to the same `o:O`

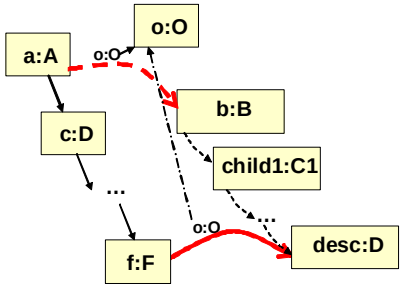




(c) Object Hierarchy: A dataflow object `o:O` has ancestors and descendants



(d) Object Reachability: `o:O` is reachable from dataflow object `c:C`



(e) Indirect Communication: `a:A` is connected to `b:B` through descendants and reachable objects.

Figure 5.1: Information content available from communication

source to a final destination through several intermediate abstract objects (Fig. 5.1(b)). This is a special case of object provenance above when the destination of the first edge is the same as the source of the second edge.

Some objects that carry protected data may not be reachable directly, but may be reachable by traversing directed edges of various types from one object to another until the abstract object containing the protected data is reached. This abstract graph can thus identify indirect communication through object reachability (Fig. 5.1(d)).

For security, it is important to identify the object that a dataflow edge refers to. Some

objects that carry protected data may be part of some other object that seems to carry only unprotected data. These cases can be identified by traversing the abstract object hierarchy. The abstract graph thus allows identifying indirect communication through descendants and ancestors. A descendant of an abstract object is the abstract object itself or a child abstract object of another descendant. Similarly, an ancestor of an abstract object is the abstract object itself or a parent of another ancestor. The abstract graph thus supports reasoning about indirect communication through the abstract object hierarchy. In particular, there are two cases: (1) an indirect communication through *ancestors* exists if there is an edge from or to an ancestor of the source or the destination; or (2) an indirect communication through *descendants* exists if there is an edge from or to a descendant of the source or the destination (Fig. 5.1(c)).

Security information that is not directly extracted from the code may be added to the abstract graph after the fact as security properties to abstract objects and edges. The set of security properties include some common properties such as trust level and confidentiality that are typically used in architectural level reasoning. Properties can be further extended for a particular system or to enable additional reasoning.

The abstract graph allows automating some of the reasoning about security-relevant architectural flaws through machine-checkable constraints on the abstract runtime structure of the system. The constraints use queries that are executed against the abstract graph. The queries can have two forms: (1) *Selection queries* return a set of abstract objects or a set of edges; and (2) *Property queries* support assigning security properties to a set of abstract objects or a set of edges. The returned sets include abstract objects or edges that satisfy a predicate. The types of predicates can be extended, e.g., is an abstract object an instance of, is an abstract object a descendant of, etc.

A constraint is a predicate on the sets returned by the queries. For example, a constraint can check if a query returns an empty set, or if the intersection of two returned sets is empty. The constraints separate the security policy from the annotations and the extraction that

shows the as-built system. While the extraction is just descriptive, the constraints can be more prescriptive and enforce security policies. The set of constraints can be extended for a particular system or reused across systems (e.g., all mobile applications have to obey a given security policy).

The rest of the section defines the Scoria process and these concepts more precisely.

5.1.1 The Scoria Process

The architects supported by Scoria find architectural flaws using the following process based on a security model named *SecGraph* (Section 5.3). The *SecGraph* augment the information on the OGraph with additional design intent. The *SecGraph* is just a wrapper of the underlying OGraph with placeholders to assign security properties to OObjects, and OEdges, and with methods for querying the graph.

1: The architects add annotations to code and typecheck them, fixing any of the typechecker warnings in the process.

2: Second, the architects use ScoriaX that automatically extracts an OGraph from the code with annotations and creates the *SecGraph*.

3: As an optional step, the architects assign security properties to objects and edges as needed using property queries on the *SecGraph*.

4: The architects write and execute constraints on the sets returned by selection queries on the *SecGraph*.

5: The architect can write additional constraints, assign more security properties or refine the annotations to fine-tune the precision of the extraction and repeat the process.

6: Finally, the architects trace to code from suspicious edges that the constraints highlight and inspect potential architectural flaws.

5.1.2 Incrementality of Scoria

The Scoria process is incremental such that the architects can begin by assigning property values to only a few of the objects. Scoria assigns the default property value `Unknown` to all the objects in the *SecGraph*. Scoria can be customized for libraries and frameworks where specific methods return confidential values, or are known to disclose information passed as arguments. The same queries can then be reused to find vulnerabilities across systems that use the same library or framework.

If Scoria finds no vulnerabilities, the architect can write application-specific queries to learn more about the system and to write additional constraints. Also, if Scoria returns too many false positives, the architect can write more specific queries to reduce their number. For example, given a set of dataflow edges, the architect can select only those edges that trace to code to an application-specific method.

5.2 Definitions

Object identity *means that every abstract object in an object graph is uniquely identified.* As a result, object identity enables the architects to assign a value to a security property of an object `o:O` and enables comparison of references. Therefore, the property value can be accessed from all the dataflow edges that refer to `o:O`. Since `o:O` is uniquely identified, the property value is the same for all these dataflow edges. Also, two distinct abstract objects `o1:C` and `o2:C` can have different property values even if `o1` and `o2` are of the same class `C`.

Object Provenance *is a query that return the set of dataflow edges* e_1 *from* `a:A` *to* `b:B` *such that another dataflow edge* e_2 *exists from* `c:C` *to* `d:D` *and both* e_1 *and* e_2 *refer to the same object* `o:O` (Fig. 5.1(a)). For security, the architects can decide if the communication of the object `o:O` is suspicious by writing a constraint that checks if the set returned by an object provenance query is empty.

Object Transitivity *is defined as the communication of an object* `o:O` *from* `a:A` *to* `c:C` *where a path of dataflow edges exists from* `a:A` *to* `c:C` *through some intermediate object* `b:B`*, where all the dataflow edges in the path refer to the same object* `o:O` (Fig. 5.1(b)). Object transitivity is important for security to reason about the initial source of a dataflow object.

In an object graph, a dataflow edge refers to an object that can have a substructure or can reach other objects.

Object Hierarchy *is defined as descendants and ancestors of an object* `o:O` *that are sets of objects such that a transitive parent-child relation exists between an object in the sets and* `o:O`. Object hierarchy is important for security because confidential information may be a descendant of a dataflow object, or a dataflow object can be part of the substructure of an object that represents confidential information (Fig. 5.1(c)).

Object reachability *is defined as a path of edges of various types from a source object* `c:C` *to a given object* `o:O` (Fig. 5.1(d)). The object `o:O` is reachable from `c:C` if a path exists from `c:C` to `o:O`. Object reachability is important for security because the object that represents confidential information can be reachable from a dataflow object. Then, the architects can consider such an edge to be suspicious. There are different types of relations between objects. In addition to dataflow, we also consider points-to and creation communication.

A points-to communication *exists from the source object* `a:A` *to the destination object* `b:B` *if one of* `a`*'s field* `f` *is a reference to* `b:B`. The label of the points-to edge is the field `f`. A points-to edge represents a persistent relation between objects and is relevant for security because a dataflow object can be used to reach an object that represents confidential information, although the dataflow object itself is not confidential.

A creation communication *exists from the source object* `a:A` *to the destination object* `b:B` *if* `a` *creates* `b` *from an object* `o:O`. Similar to a dataflow edge, a creation edge refers to an object, and is relevant for security it may lead to vulnerabilities such as information disclosure or tampering.

Architects can obtain additional information about objects that are indirectly connected by a dataflow communication through descendants or reachable objects.

Indirect communication *exists from a source object* `a:A` *to a destination object* `b:B` *if a dataflow or a creation edge exists from a descendant of* `a` *or an object reachable from* `a` *to a descendant of* `b` *or an object reachable from* `b`. Indirect communication is relevant for security because a confidential object may be passed to a descendant of an untrusted object and the communication may not be explicit in the abstract graph.

5.3 Security Graph and Queries

To find security vulnerabilities, one strategy is to conduct ARA from the attackers point of view considering the following threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege (STRIDE) [60]. The constraints in Scoria focus on vulnerabilities related to dataflow communication such as Spoofing, Tampering and Information Disclosure. Finding flaws, such as Denial of Service, requires tracking changes in the state of objects, which the extracted object graph, focused on structural information, does not support.

In previous work [7, 8], some of the STRIDE checks were automated by defining component-level properties such as *TrustLevel* represented using an Architecture Description Language. This work largely reuses the same model but represents it on a lightweight adapter of the OGraph, the *SecGraph*. This section describes the data types of the *SecGraph* and introduces formally examples of selection queries and property queries.

In the implementation of Scoria, the *SecGraph* is a library that provides several interfaces and default implementations of selection queries and property queries. Thus, Scoria provides extensibility and allows architects to write constraints in a Java-like language, and uses assertions similar to the ones used in unit tests.

SecGraph
- *root: SecObject*
- *edges(): EdgeSet*
- *objects(): ObjSet*
- *objectProvenance(...): EdgeSet*
- *objProvenanceIndirect(...): EdgeSet*
- *connectedByDF(...): EdgeSet*
- *getFlowToSink(...): EdgeSet*
- *getEdgesByFlow(...): EdgeSet*
- *reachables(o:SecObject, eTypes:Set⟨EdgeType⟩): ObjSet*

ObjSet `extends` *Set⟨SecObject⟩*
- *name:String*
- *prop: Property*

EdgeSet `extends` *Set⟨SecEdge⟩*
- *name:String*
- *prop: Property*

SecObject `extends` *SecElement*
- *C: Type*
- *parentObj:SecObject*
- *parentDom:SecDomain*
- *descendants() : ObjSet*
- *ancestors() : ObjSet*

SecDomain `extends` *SecElement*
- *d: String*

SecEdge `extends` *SecElement*
- *src: SecObject*
- *dst: SecObject*
- *edgeType: EdgeType*

EdgeType:`DataFlow` |`PointsTo` |`Creation`

DataFlowEdge `extends` *SecEdge*
- *flow: SecObject*
- *direction:* `Import` |`Export`

PointsToEdge `extends` *SecEdge*
- *fieldName: String*

CreationEdge `extends` *SecEdge*
- *flow: SecObject*

SecElement
- *name: String*
- *traceability(): Set⟨Traceability⟩*
- *trustLevel:* `Trusted` |`Untrusted` |`Unknown`
- *isConfidential:* `True` |`False` |`Unknown`
- *isEncrypted:* `True` |`False` |`Unknown`
- *isSerialized:* `True` |`False` |`Unknown`
- *isSanitized:* `True` |`False` |`Unknown`
- *isTransient:* `True` |`False` |`Unknown`

Traceability
- *expression: AstNode*

ClassInstanceCreation extends AstNode

Figure 5.2: Partial representation of the *SecGraph*. Continued in Fig. 5.3,5.4

5.3.1 Data Types

A *SecGraph* is as an OGraph enriched with queries and security properties. The *SecGraph* has *SecObject*s and *SecDomain*s as nodes, and *DataFlowEdge*s, *PointsToEdge*s and *CreationEdge*s as edges (Fig. 5.2). The *SecGraph* provides queries such as *descendants()* for finding the set of descendants of a given *SecObject*, and *reachables()* for getting the set of reachable *SecObject* from a given *SecObject* using various types of edges. In the *SecGraph*, I omit the details of the static analysis that extracts the OGraph (Chapter 3, page 57), and focus on the queries and constraints. The queries return an *ObjSet* and *EdgeSet* that have a name and property values for their elements. An *ObjSet* is cross-cutting to a *SecDomain* where an *ObjSet* contains any *SecObject* that satisfy a predicate (a *Condition*). An *ObjSet* is different from a *SecDomain* such that the same *SecObject* can be part of two *ObjSet*s, but the same *SecObject* cannot be part of two *SecDomain*s.

All these data types extend the base type *SecElement*, which has traceability information

Query based on object provenance

$objectProvenance(fSrc : SecObject, fDst : SecObject,$ $sSrc : SecObject, sDst : SecObject) : EdgeSet$
$objectProvenance := \bigcup\{edge\}$ s.t. $edge : DataFlowEdge \in edges() \wedge$ $\exists o \in objects()$ s.t. $o \in flow(connectedByDF(fSrc, fDst)) \wedge$ $edge \in connectedByDF(sSrc, sDst) \wedge edge.flow = o$
$flow(dfEdges : Set\langle DataFlowEdge\rangle) : ObjSet$
$flow := \bigcup\{obj\}$ s.t. $obj : SecObject \in objects() \wedge$ $\exists edge \in dfEdges$ s.t. $edge.flow = obj$
$connectedByDF(src : SecObject, dst : SecObject) : EdgeSet$
$connectedByDF := \bigcup\{edge\}$ s.t. $edge : DataFlowEdge \in edges() \wedge$ $(edge.src = src \ \wedge \ edge.dst = dst)$
$objProvenanceIndirect(fSrc : SecObject, fDst : SecObject,$ $sSrc : SecObject, sDst : SecObject) : EdgeSet$
$objProvenanceIndirect := \bigcup\{edge\}$ s.t. $edge{:}DataFlowEdge \in edges() \wedge$ $\exists o \in objects()$ s.t. $o \in flow(connectedIndirect(fSrc, fDst)) \wedge$ $edge \in connectedByDescendants(sSrc, sDst) \vee$ $edge \in connectedByReachablility(sSrc, sDst) \wedge edge.flow = o$

Figure 5.3: Object provenance queries on a *SecGraph*

consisting of a set of traceability entries that correspond to expressions in the code. A traceability entry is a reference to a node in the Abstract Syntax Tree (AST) such as a *ClassInstanceCreation*. One advantage of using AST[2] in the representation of *Traceability* instead of just a file name and a line number is that the constraints can include information about code structure such as a name of a method. Another advantage of using an AST is that Architects can refine annotations, which may be added on new lines without breaking the traceability links.

5.3.2 Selection Queries

The simplest selection queries use methods specified on an instance of the *SecGraph*, *G*. For example, *G.edges()* returns all the edges in the graph. An instance *edge* of type *DataFlowEdge*, *edge.flow* returns the dataflow object that *edge* refers to. A query can also invoke methods in complex expressions such as *edge.flow.descendants()*, which returns the

[2]The tool converts from the intermediate AST representation to the AST used by the IDE.

Assign property values and select objects

$setObjectsProperty(props : Set\langle \text{Property} \rangle, cond : Condition)$
$\forall obj \in objects(),$ s.t. $cond.satisfiedBy(obj) \Longrightarrow$
$\exists p : IsConfidential \in props \ \wedge \ obj.isConfidential := p \ \vee$
$\exists p : TrustLevel \in props \ \wedge \ obj.TrustLevel := p \ \vee \ldots$

$getObjectsByCond(cond : Condition\ ) : ObjSet$
$getObjectsByCond := \bigcup\{obj\}$ s.t. $obj : SecObject \in objects() \wedge$
$\quad cond.satisfiedBy(obj)$

Select edges based on security properties

$getFlowToSink(fProps : Set\langle \text{Property} \rangle, sProps : Set\langle \text{Property} \rangle) : EdgeSet$
$flows := getObjectsByCond(fProps)\ \ sinks := getObjectsByCond(sProps)$
$getFlowToSink := getFlowToSink(flows,sinks)$

$getFlowToSink(flows : ObjSet, sinks : ObjSet) : EdgeSet$
$getFlowToSink := \bigcup\{e\}$ s.t. $e \in edges() \wedge$
$e : DataFlowEdge \ \vee \ e : CreationEdge \wedge$
$\exists sink : SecObject \in sinks \ \wedge \ \exists o : SecObject \in objects()$ s.t.
$e \in connectedByDescendants(o, sink) \vee$
$e \in connectedByReachability(o, sink, \{\texttt{PointsTo}\}) \wedge$
$flw \in flows \cap (e.flow.descendants() \cup reachables(e.flow, \{\texttt{PointsTo}\}))$

Conditions based on type+object hierarchy

$InstanceOf(objT : Type)$
$satisfiedBy(obj : SecObject) := obj.C.isSubTypeOf(objT)$

$IsInDomain(prntT : Type, prntD : String, objT : Type)$
$satisfiedBy(obj : SecObject) := obj.C.isSubTypeOf(objT) \wedge$
$obj.parentDom.d = prntD \ \wedge \ obj.parentObj.C.isSubTypeOf(prntT)$

$IsChildOf(prntO : SecObject, prntD : String, objT : Type)$
$satisfiedBy(obj : SecObject) := obj.C.isSubTypeOf(objT) \wedge$
$obj.parentDom.d = prntD \ \wedge \ obj.parentObj = prntO$

Condition based on type+object reachability

$IsInstOfRchblFromInstOf(st : Type, dt : Type, eType : Set\langle EdgeType \rangle)$
$isSrc := InstanceOf(st)\ \ isDst := InstanceOf(dt)$
$satisfiedBy(obj : SecObject) := \exists\ obj \in getObjectsByCond(isSrc) \wedge$
$\exists\ dobj \in getObjectsByCond(isDst) \ \wedge \ obj \in reachables(dobj, eType)$

Condition based on type+object traceability

$IsCreateadInMD(methDecls : Set\langle \texttt{MethodDeclaration} \rangle)$
$satisfiedBy(obj : SecObject) :=$
$\quad \exists cic : \texttt{ClassInstanceCreation} \in obj.traceability()$ s.t.
$\quad md : \texttt{MethodDeclaration} \in cic.declarations() \ \wedge \ md \in methDecls$

Figure 5.4: Selection and property queries on a *SecGraph*

set of all descendants of the dataflow object.

The query *objectProvenance* returns a set of dataflow edges such that the same object that flows from a first source to a first destination also flows from a second source to a second destination (Fig. 5.3). The query uses *connectedByDF()* that returns the set of dataflow edges between two objects, and *flow()* that returns the set of dataflow objects that the dataflow edges in a given set refer to. Reasoning about object provenance may also involve indirect communication through descendants and reachable objects. The query *objProvenanceIndirect* uses *connectedByDescendants()* that returns the set of dataflow edges from a descendant of the source to a descendant of the destination, and *connectedByReachability()* that return the set of dataflow edges from an object reachable from the source to an object reachable from the destination.

The *SecGraph* also provides the selection query *getFlowToSink* that takes as arguments two sets of property values *fProps* and *sProps* and returns those dataflow edges or creation edges that refer to a *SecObject* with the property values *fProps* and has as destination a *SecObject* with the property values *sProps*. The query *getFlowToSink* has flags to control whether to chase descendant and reachable *SecObject*s, as well as indirect communication, such that queries are general expressed only in terms of security property.

A selection query such as *getObjectsByCond()* returns the *ObjSet* or *EdgeSet* that satisfy a *Condition*. The *SecGraph* has several *Condition*s that can be further extended. A constraint is a predicate on the returned *ObjSet* or *EdgeSet* such as an *ObjSet* is empty, the intersection or union of two *ObjSet* is empty.

5.3.3 Conditions

A *Condition* is a predicate that takes as arguments elements of the object structure, code structure, or property values. The predicate is implemented by the *satisfiedBy* method that is invoked for every element as the query traverses the *SecGraph*. For example, a selection query using *IsCreatedInMD* returns the set of *SecObject*s instantiated in the body of one

of the methods provided as argument. A query based on type only is less precise than a query that combines information from the code structure and the *SecGraph*. For example, a query using the condition *InstanceOf* returns the set of all *SecObject*s of a type specified as argument. Sometimes, not all objects of a type should be treated the same. Using the condition *IsInDomain*, a selection query returns a subset of these *SecObject*s by specifying the type of the parent object, and name of a parent domain. In addition, a *Condition* can use reachability information. For example a selection query based on the condition *IsInstOfRchblFromInstOf* returns the set of *SecObject*s of a type *dt* that are reachable from *SecObject*s of type *st* through various type of edges.

5.3.4 Property Queries

A property query such as *setObjectsProperty()* assigns security properties to a set of objects or a set of edges (Fig. 5.4). To allow architects to specify design information unavailable in the code, a *SecElement* has an extensible set of security properties. Each *Property* includes predefined values and the value `Unknown` used for default initialization. For example, the architects can specify objects that are trusted, or confidential. The property values are stored at the level of the *SecGraph*. In the rest of the thesis, when we say we assign to an object *TrustLevel*.`True`, it means the we assign to the object the security property *TrustLevel* with the value `True`.

5.3.5 Extensibility

For extensibility, *Condition* and *Property* are defined as interfaces, and several concrete implementations of these interfaces are described in this chapter. By design, the queries have parameters declared in terms of these interfaces. The architects can define additional conditions by defining classes that implement the *Condition* interface, which may use application-specific information such as name of domains, types and methods that the architects learn about by inspecting the *SecGraph* or the code.

5.4 Summary

This chapter describes Scoria and shows that it meets the extensibility requirement (**SC3**) using an incremental process to find security vulnerabilities. Scoria is based on the security model *SecGraph* that supports queries in terms of object provenance (*H2a*) and indirect communication (*H2f*). Chapter 6 describes an evaluation Scoria providing quantitative and qualitative support for discussing the recall and the precision of Scoria.

Chapter 6 Finding Architectural Flaws

This chapter[1] presents the evaluation of Scoria on finding security vulnerabilities and provides support for the hypothesis H2 (Section 1.5.1, page 9).

> *Based on a sound, hierarchical, abstract object graph with dataflow edges that refer to objects, the architect writes machine-checkable constraints in terms of one or more of the Scoria features to effectively (high recall and precision) to detect security vulnerabilities.*

This chapter is organized as follows. First, to show that Scoria meets the requirement (**SC2**) and supports automation, Section 6.1 presents machine-checkable constraints executed on small examples with injected vulnerabilities. The examples are based on the non-compliant code from the CERT Oracle Secure Coding Standard for Java [81] for which automated detection was previously unavailable. Section 6.2 shows that constraints are extensible (**SC3**) and can find information disclosure in an open-source Android application. To estimate the recall and precision, Scoria is evaluated on a benchmark of test cases designed by other researchers [48] that focuses on information disclosure in Android apps (Section 6.3). Section 6.5 summarizes the chapter and concludes.

6.1 Evaluation on CERT Rules

This section provides evidence that Scoria supports automation of informal security constraints using five extended examples based on the CERT rules. The CERT coding standards are the result of security communities effort to support developers that use programming languages such as Java, C/C++, and Perl on writing code without security vulnerabilities by following best-practices rules. I focus on the rules for Java [81], where each CERT rule has an informal description, examples of compliant and non-compliant code, and available

[1] The evaluation on CERT and UPMA appeared in [133]. A tool demonstration of Scoria appeared in [132]

solutions for automated detection. The CERT rules are informally described and for most rules automated detection is unavailable.

6.1.1 Constraints that Implement CERT Rules

Selection of CERT rules. To select the five CERT rules, I used the following criteria. First, the selected rules are related to information disclosure or tampering. Second, the rules involves communication of objects, and third, the automated detection of the rule was previously unavailable. Other CERT rules, not selected, require reasoning about control flow or object state or involve the denial of service or elevation of privileges vulnerabilities.

I created complete examples from the non-compliant code fragments, then guided by the informal description, I wrote constraints to trigger the automated detection of the architectural flaws. Since the CERT rules are general, the machine-checkable constraints could conceivably be reused across Java applications with minor changes.

MSC00-J. Use `SSLSocket` rather than `Socket` for secure data exchange.

A client-server application that highlights a potential information disclosure if the communication between the server and the client is not encrypted. The example consists of two objects, `client:EchoClient` and `server:EchoServer` that communicate via a message `ss:String` received from user input. The communication is established via a `sckt:Socket` object, which contains two streams of data, namely, `in:InputStream` and `out:OutputStream`. The object `sckt:Socket` does not provide encryption and should be used only if the data transmitted is not confidential or if the network is trusted. Otherwise, the implementation should use encryption protocols such as Secure Sockets Layer (SSL) to ensure that the channel is not vulnerable. The server supports both types of communication and has an object `sslSckt:SSLSocket` that encrypts data.

To find a possible information disclosure, the constraint *insecureDataflows* (line 6.1) uses object provenance and checks if the same object that flows from an object of type

$$insecureDataflows(g : SecGraph, isSrc : Condition, isDst : Condition, isUntrusted : Condition) \quad (6.1)$$

$$insecureDataflows := \bigcup\{e\} \text{ s.t. } e : DataFlowEdge \in g.edges() \wedge \quad (6.2)$$

$$e \in g.objProvenanceIndirect(inObj, tObj, tObj, uObj) \text{ where} \quad (6.3)$$

$$inObj \in g.getObjectsByCondition(isSrc) \quad (6.4)$$

$$tObj \in g.getObjectsByCondition(isDst) \quad (6.5)$$

$$uObj \in g.getObjectsByCondition(isUntrusted) \quad (6.6)$$

$$isSrc := IsInDomain(\texttt{Main}, \texttt{DATA}, \texttt{InputStream}) \quad (6.7)$$

$$isDst := InstanceOf(\texttt{EchoClient}) \cup InstanceOf(\texttt{EchoServer}) \quad (6.8)$$

$$isUntrusted := InstanceOf(\texttt{Socket}) \quad (6.9)$$

$$\{e1, e2\} \subseteq insecureDataflows(G, isSrc, isDst, isUntrusted) \quad (6.10)$$

$$e1 := \langle \texttt{client:EchoClient}, \texttt{out:PrintWriter}, \texttt{ss:String} \rangle \quad (6.11)$$

$$e2 := \langle \texttt{server:EchoServer}, \texttt{wrtr:OutputStream}, \texttt{ss:String} \rangle \quad (6.12)$$

```
1 class EchoClient {
2   void run(){
3     Socket sckt = new Socket("localhost", 9999);
4     OutputStream out = sckt.getOutputStream();
5     PrintWriter out = new PrintWriter(out, true);
6     while ((userInput = System.in.read()) != null) {
7       out.write(userInput); //inf. disclosure
8       System.out.println(userInput) //false positive
9     }
10 }
```

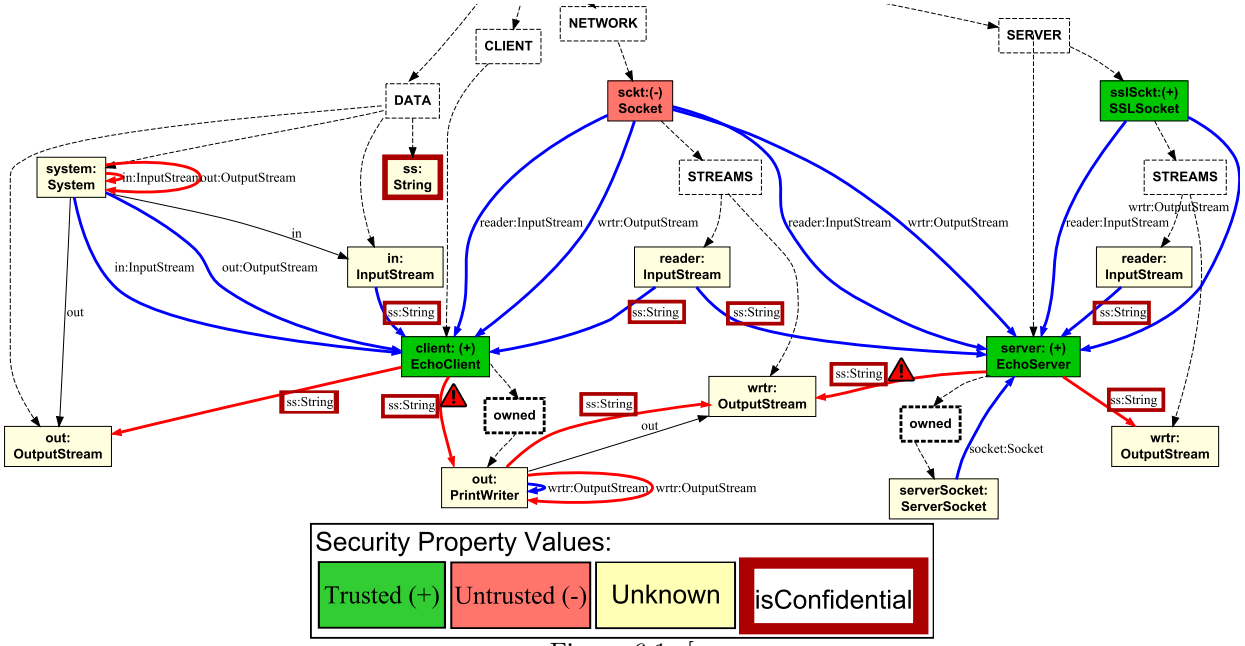


Figure 6.1: [Constraint, code fragments and SecGraph for the CERT rule MSC00-J] Constraint, code fragments and SecGraph for the CERT rule MSC00-J. Confidential information flows to a descendant of `sckt:Socket`.

`InputStream` in the domain `DATA` of the class `Main` (line 6.7) to an object of type `EchoClient` also flows to an object `sckt:Socket`. The constraint also uses object provenance (line 6.3) and checks that the object flows to the descendants or reachable objects of `sckt:Socket`. Indeed, the same object `ss:String` flows from `client:EchoClient` to the object `out:PrintWriter` from which a descendant of `sckt:Socket` is reachable (line 6.11). The object `ss` also flows from `server:EchoServer` to a descendant of `sckt:Socket` (line 6.12). The constraint highlights these two dataflow edges as suspicious. From a suspicious dataflow edge, the architects can trace to the method invocation `out.write(stdInput)` (line 7 in the code fragments). There are no coding defects, however an information disclosure does exist.

If the constraint were to use the type information only and consider all objects of type `OutputStream` as untrusted, it would report four suspicious dataflow edges including two false positives: the dataflow edge from `client` to the standard output stream, and the dataflow edge from `server` to a child of `sslSckt:SSLSocket`.

FIO05-J. Do not expose buffers created using the wrap() or duplicate() methods to untrusted code.

The class `CharBuffer` has methods that wrap an array of primitive types into an object of type `CharBuffer`. According to the Java documentation, modifications of such an object cause the array to be modified and vice-versa [81, 98]. Therefore, passing `cb:CharBuffer` to an untrusted object exposes the array that is reachable from `cb`. Instead, an object with a reference to a copy of the array should be passed.

The constraint *bufferExposures* (line 6.13) selects the objects of type `CharBuffer` created in the body of the method `wrap` or duplicate (line 6.17). Next, the constraint selects the children of these objects that are of type `char[]` (line 6.19) and assigns to them *IsConfidential*.`True` (line 6.20). The constraint also assigns *TrustLevel*.`Untrusted` to `c:Client`, which represents untrusted code (line 6.21). The query *getFlowToSink* returns a suspicious edge

$$bufferExposures(g : SecGraph) \tag{6.13}$$

$$bufferExposures := g.getFlowToSink(\{IsConfidential.\texttt{True}\}, \{TrustLevel.\texttt{Untrusted}\}) \tag{6.14}$$

$$mdWrap = G.getMethodDecl(\texttt{CharBuffer.wrap}) \tag{6.15}$$

$$mdDuplicate = G.getMethodDecl(\texttt{CharBuffer.duplicate}) \tag{6.16}$$

$$isInMD := IsCreatedInMD(\{mdWrap, mdDuplicate\}) \tag{6.17}$$

$$\forall o \in getObjectsByCond(isInMD), \tag{6.18}$$

$$\quad isChild := IsChildOf(o, \texttt{OWNED}, \texttt{char[]}) \tag{6.19}$$

$$\quad G.setObjectsProperty(IsConfidential.\texttt{True}, isChild) \tag{6.20}$$

$$\quad G.setObjectsProperty(TrustLevel.\texttt{Untrusted}, InstanceOf(\texttt{Client})) \tag{6.21}$$

$$\{e\} \subseteq bufferExposures(G) \tag{6.22}$$

$$e := \langle \texttt{m:Main}, \texttt{c:Client}, \texttt{cb:CharBuffer} \rangle \tag{6.23}$$

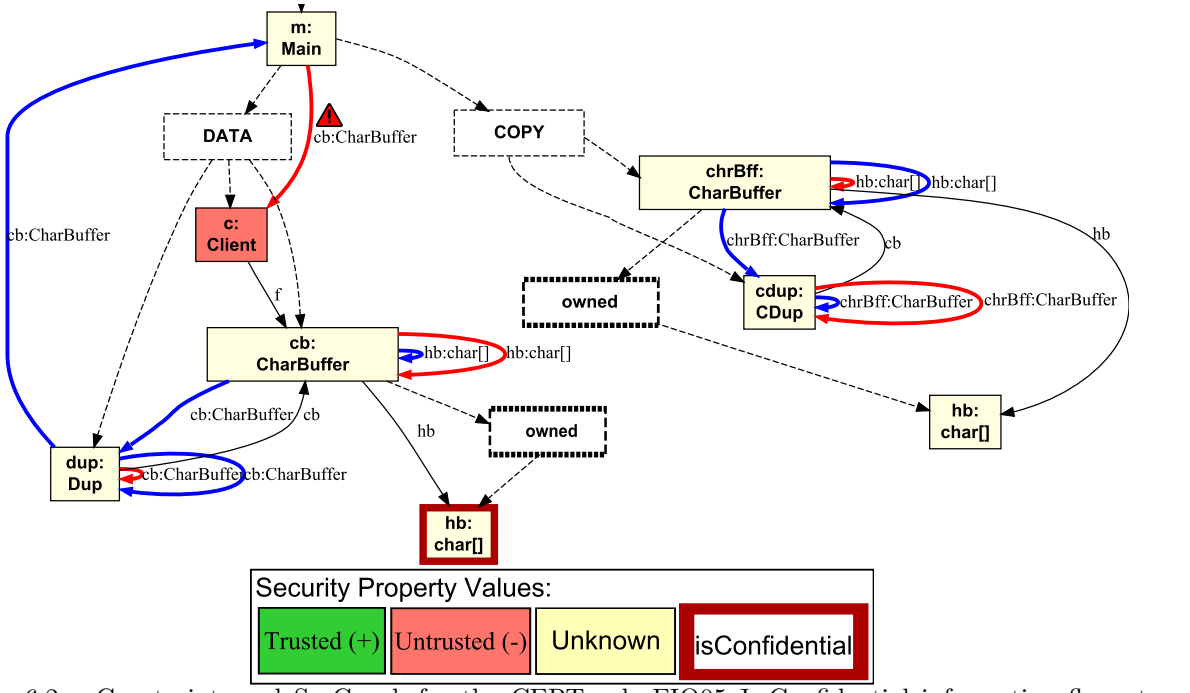


Figure 6.2: Constraint, and SecGraph for the CERT rule FIO05-J. Confidential information flows to a malicious client.

(line 6.23) because the dataflow object `cb:CharBuffer` has a points-to edge to the confidential object of type `char[]` and the destination object is `c`. This constraint may also consider objects of types that are similar to `CharBuffer` such as `IntBuffer`.

SER03-J. Do not serialize unencrypted, sensitive data.

Serialization saves an object persistently. All the fields reachable from the object are also saved unless the developers mark them as transient. An information disclosure exists if a

confidential, unencrypted object is serialized and if the serialized data is not itself stored securely.

The CERT example [81, SER03-J] uses the `Serializable` interface. Since other libraries can also be used for serialization, the *SecGraph* has the property *IsTransient* to keep the constraint independent from a specific library. The constraint *insecureSerializations* (line 6.24) assigns *TrustLevel*.`Untrusted` to objects of type `ObjectOutputStream` (line 6.29), then assigns *IsConfidential*.`True` and *IsEncrypted*.`False` to objects of type `Double` in the domain `OWNED` of objects of type `Point` (line 6.31). Next, it assigns *IsTransient*.`False` to points-to edges from objects of type `Point` to objects of type `Double` (line 6.33). Finally, *insecureSerializations* checks if the *EdgeSet* returned by *getFlowToSink* is not empty such that the outgoing points-to edge from the dataflow object has *IsTransient*.`False`.

The constraint finds an information disclosure as a dataflow from `coord` to `oout` (line 6.35), where the dataflow object is `p:Point` that has references to `value:Double`. The object `value:Double` has the property *IsConfidential*.`True` and should not be serialized. However, the points-to edge from `p` to `value` has *IsTransient*.`False`, and serialization occurs. Next, the architects can trace to code and find the method invocation `oout.writeObject(p)` in the body of `Coordinates.run`.

By using the condition *IsInDomain*, the constraint assigns property values only to a subset of all the objects of type `Double`, i.e., it does not include children of objects of type `Coordinate`. If the constraint were to use the *InstanceOf* condition instead of *IsInDomain*, all the objects of type `Double` would be considered confidential, and *insecureSerializations* would return one false positive.

FIO13-J. Do not log sensitive information outside a trust boundary.

Logging enables a program to collect essential information for debugging. However, logging confidential information such as IP addresses may be prohibited by law in many countries [81, FIO13-J]. The class `Logger` in Java provides basic functionality for logging.

$$insecureSerializations(g : SecGraph) \tag{6.24}$$

$$insecureSerializations := \bigcup\{e\}\ e : DataflowEdge \wedge \tag{6.25}$$

$$e \in g.getFlowToSink(\{IsConfidential.\texttt{True}, IsEncrypted.\texttt{False}\}, \{TrustLevel.\texttt{Untrusted}\})\ \text{s.t.} \tag{6.26}$$

$$\exists pte \in g.getOutEdges(e.flow(), \texttt{PointsTo}) \ \wedge \ pte.isTransient = \texttt{False} \tag{6.27}$$

$$isostream := InstanceOf(\texttt{ObjectOutputStream}) \tag{6.28}$$

$$G.setObjectsProperty(TrustLevel.\texttt{Untrusted}, isostream) \tag{6.29}$$

$$points := IsInDomain(\texttt{Coordinate}, \texttt{OWNED}, \texttt{Point}) \tag{6.30}$$

$$isXY := IsInDomain(\texttt{Point}, \texttt{OWNED}, \texttt{Double}) \tag{6.31}$$

$$G.setObjectsProperty(\{IsConfidential.\texttt{True}, IsEncrypted.\texttt{False}\}, isXY) \tag{6.32}$$

$$G.setEdgesProperty(IsTransient.\texttt{False}, IsPtEdge(points, isXY)) \tag{6.33}$$

$$\{e\} \subseteq insecureSerializations(G) \tag{6.34}$$

$$e = \langle \texttt{coord:Coordinates}, \texttt{oout:ObjectOutputStream}, \texttt{p:Point} \rangle \tag{6.35}$$

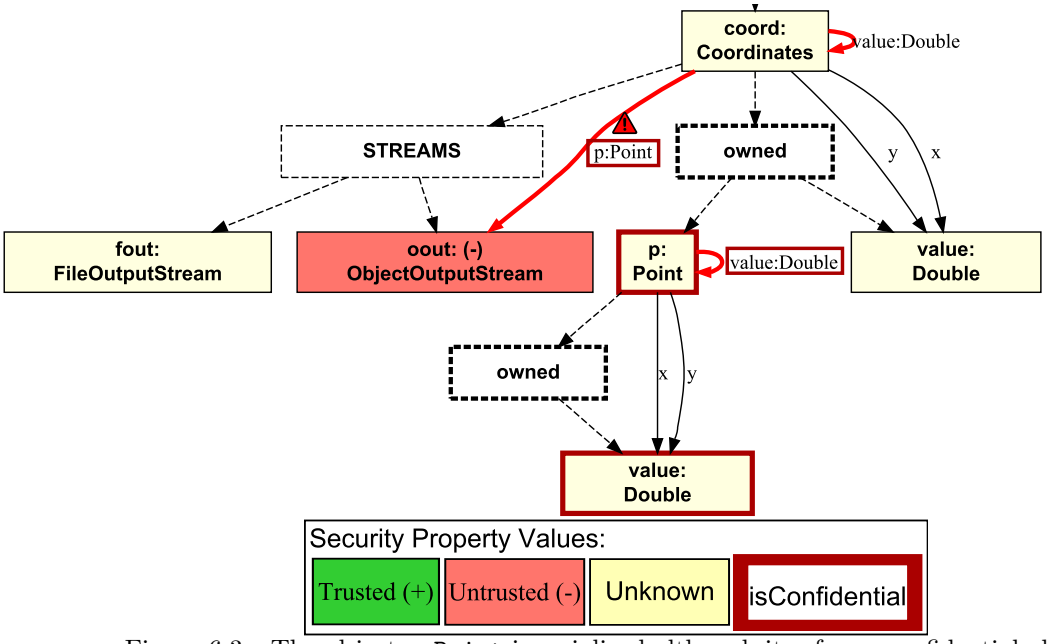


Figure 6.3: The object `p:Point` is serialized although it refers a confidential object.

The constraint *insecureLogging* (line 6.36) checks if dataflow edges exist such that a dataflow object has *IsConfidential*.`True` and the destination has *TrustLevel*.`Untrusted`. The constraint uses reachability information to assign *IsConfidential*.`True` only to the objects of type `String` reachable from an object of type `InetAddress` (line 6.39). The constraint highlights a dataflow edge from `srvr` to `log` that refers to the object `ip:String`. The architects can trace to the line of code `logger.severe(machine.getHostAddress())`, which indeed logs the IP address (line 6.43 Fig. 6.4).

$$insecureLogging(g : SecGraph) \tag{6.36}$$

$$insecureLogging := g.getFlowToSink(IsConfidential.\texttt{True}, TrustLevel.\texttt{Untrusted}) \tag{6.37}$$

$$rch := IsInstOfRchblFromInstOf(\texttt{InetAddress}, \texttt{String}, \{\texttt{PointsTo}\}) \tag{6.38}$$

$$G.setObjectsProperty(IsConfidential.\texttt{True}, rch) \tag{6.39}$$

$$isLog := InstanceOf(\texttt{Logger}) \tag{6.40}$$

$$G.setObjectsProperty(TrustLevel.\texttt{Untrusted}, isLog) \tag{6.41}$$

$$\{e\} \subseteq insecureLogging(G) \tag{6.42}$$

$$e := \langle \texttt{srvr:LogServer}, \texttt{log:Logger}, \texttt{ip:String} \rangle \tag{6.43}$$

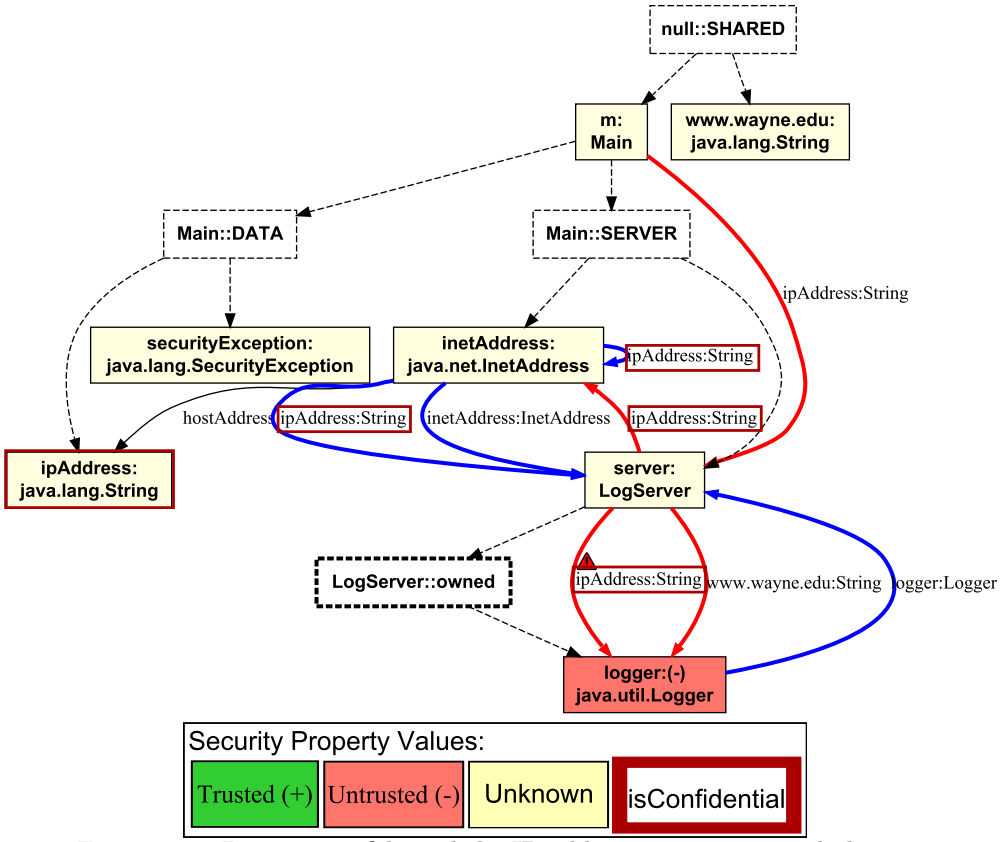


Figure 6.4: Logging confidential the IP address to an untrusted object.

To assign the *IsConfidential* property, the architects use a condition that considers type and reachability information. If the architects were to use a condition that considers only the type information, *insecureLogging* would return a false positive for the expression `logger.getLogger(name)`, where the `name` argument does not refer to a confidential object.

IDS07-J. Do not pass untrusted, unsanitized data to the Runtime.exec() method.

A Java program can invoke external commands provided by the operating system such as a command that lists a folder's content. An architectural flaw exists if an attacker can tamper with the input and launch arbitrary injected commands with the privileges of the target user. As a solution, the program must sanitize untrusted objects flowing along untrusted communication [81, IDS07-J].

An attacker can invoke arbitrary commands on Windows by tampering with the arguments of the method `Runtime.exec`. The constraint *commandInjections* (line 6.44) finds tampering by using object provenance, and traceability information. It uses object provenance to check if the same object that is passed as an argument to `m:Main` is also passed to an object of type `Runtime` (line 6.46). It also uses traceability (line 6.50) and highlights only dataflow edges due to the method invocation `Runtime.exec` (line 6.56) (Fig. 6.5). The architects trace to code and find the `rt.exec("cmd.exe /C dir " + dir)`. If the constraint were to use only information about the type of the arguments, it would also return a false positive for `rt.load(filename)`.

6.1.2 Results

The results show that constraints using all Scoria features can detect security vulnerabilities that are architectural flaws. Two of the constraints use object provenance and do not require security properties. For the others, the architects need to assign security properties to only a few of the *SecElement*s and use the defaults elsewhere (Table 6.1). The constraints use the information provided by points-to and dataflow edges in the *SecGraph*. While all the constraints highlight dataflow edges, three of them use reachability information through points-to edges.

The evaluation highlights why Architectural Risk Analysis (ARA) requires a runtime

$$comInjections(g : SecGraph, isUntrusted : Condition, isTrusted : Condition, md : MethodDeclaration) \quad (6.44)$$

$$commandInjections := \bigcup\{e\} \text{ s.t. } e : DataFlowEdge \wedge \quad (6.45)$$

$$\exists e \in g.objectProvenance(main, o, o, rt) \text{ where} \quad (6.46)$$

$$(\exists\ main : SecObject \in g.getObjectsByCond(isUntrusted) \wedge \quad (6.47)$$

$$\exists\ rt : SecObject \in g.getObjectsByCond(isTrusted) \wedge \quad (6.48)$$

$$\exists\ o : SecObject \in g.objects()) \wedge \quad (6.49)$$

$$mInvk : MethodInvocation \in edge.traceability() \wedge \quad (6.50)$$

$$mInvk.methodDecl = md \ \wedge \ edge.isExport()) \quad (6.51)$$

$$mdExec := G.getMethodDecl(\texttt{Runtime.exec}) \quad (6.52)$$

$$isUntrusted := InstanceOf(\texttt{Main}) \quad (6.53)$$

$$isTrusted := InstanceOf(\texttt{Runtime}) \quad (6.54)$$

$$\{e\} \subseteq comInjections(G, isUntrusted, isTrusted, mdExec) \quad (6.55)$$

$$e := \langle \texttt{dirList:DirList}, \texttt{rt:Runtime}, \texttt{command:String} \rangle \quad (6.56)$$

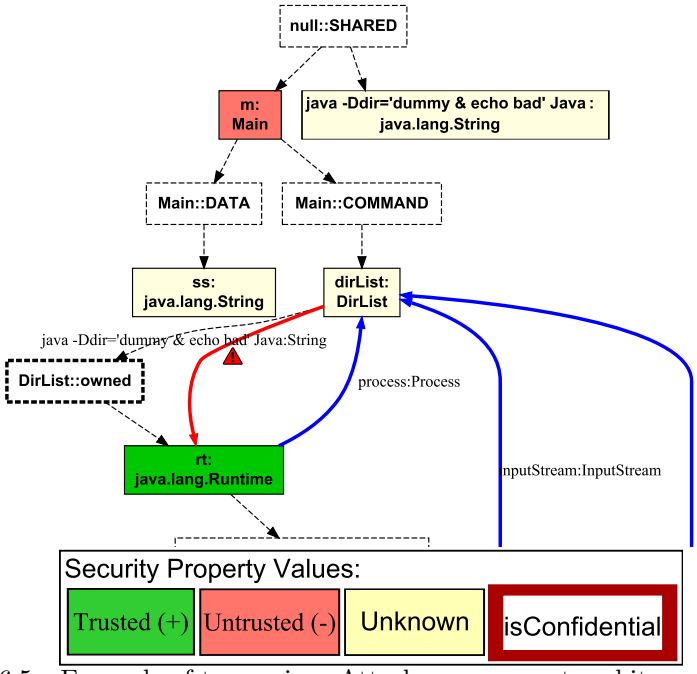


Figure 6.5: Example of tampering. Attacker can execute arbitrary commands

architecture—not a code architecture. For example, *bufferExposures* distinguishes between different objects of the same type with different property values such as the two *SecObjects* of type `char[]` and the two *SecObjects* of type `CharBuffer`. By using the runtime architecture, the constraints return fewer false positives. Some constraints also use information from the code architecture such as subtyping information or method names.

In related work [7], object graphs with points-to edges were abstracted into a runtime

Table 6.1: Summary of constraints that implement rules in the CERT Oracle Secure Coding Standard [81]

CERT-ID	vulnerability	(a) object provenance	(b) object transitivity	(c) object hierarchy	(d) object reachability	(e) object traceability	(f) indirect communication	suspicious edges	avoided false positives	(g) object security properties	(g) edge security properties
MSC00-J	information disclosure	✓	✓	✓	✓		✓	2	2	0	0
FIO05-J	information disclosure			✓	✓	✓		1	1	2	0
SER03-J	information disclosure			✓				1	1	2	2
FIO13-J	information disclosure				✓			1	1	2	0
IDS07-J	tampering	✓	✓			✓		1	1	0	0

architecture represented in an architectural description language. Constraints that check information disclosure were expressed as predicates that check only the presence or the absence of communication, which is not enough. In Scoria, not only can architects check the presence or absence of communication, they can also express constraints about the information content of the communication between objects because a dataflow edge refers to an abstract object. For example, in the constraint *insecureLogging*, the *SecGraph* has two dataflow edges from `srvr:LogServer` to `log:Logger`. If the constraint were to check only for the absence of communication, it would be too restrictive. The communication from `srvr` to `log` is required, otherwise the system becomes vulnerable to other security vulnerabilities such as repudiation [60]. Only one dataflow edge leads to information disclosure, and the constraint can tell the two edges apart by tracking the object flowing along the edge and using object provenance.

6.2 Evaluation on Open-Source Application

Previous examples presented constraints that found injected architectural flaws. Scoria can also find architectural flaws that the architects are not aware of in open-source applications. For example, an Android application is written in Java and consists of sev-

eral `Activity` components that interact with the user using a `View`. Two objects of type `Activity` communicate with each other using an Intent that represents a message used for inter- and intra-application communication.

The subject system is Universal Password Manager Android application (UPMA) version 1.13, a 4KLOC open-source Android app that encrypts and stores usernames, passwords and confidential notes in a database protected by a master password. A security vulnerability in UPMA would impact a large number of users interested in protecting their confidential information.

6.2.1 Collecting Security Constraints from Documentation

This subsection provides examples of informal constraints considered during the UPMA case study. During ARA, architects gather information about the application and informal security constraints from the existing documentation that can be application-specific or documentation of frameworks and libraries used by the application. Detailed security policies are often available in the description of standard security protocols that the application or the library uses. The architects may read and understand these security policies which are a collection of constraints and identify risks. However, since these constraints are only informally described, checking them on the implementation is challenging.

Informal constraints reusable across frameworks and libraries

Some constraints are general across applications that use the same framework or library. For example, the following security policy is documented both by researchers [28] and in the Android documentation [19]:

> *"Do not put sensitive data into Intents used to start Activities [...] applications with the `GET_TASKS` permission are able to see `ActivityManager.RecentTaskInformation` which includes the base Intent used to start Activities".*

Other security constraints are available as best-practices specific to mobile applications [2]. The following informal constraint[2] states that the clipboard can be accessed by any application installed on the device, and the copy/paste feature should be disabled in the presence of confidential information.

> *Android also supports copy/paste by default and the clipboard can be accessed by any application. Where appropriate, disable copy/paste for areas handling sensitive data. Eliminating the option to copy can help avoid data exposure.*

Informal constraints are also available in the documentation of services. Supporting third-party services require the application to ensure that the authentication credentials of these services are securely stored and communicated. For example, UPMA has support for the Dropbox service to allow users to store and share the database on several devices. The following are snippets of informal security constraints from the Dropbox documentation[3].

- *[...] all requests are done over SSL.*
- *OAuth 1.0 continues to be supported for all API requests, but OAuth 2.0 is now preferred*
- *Step 1 of authentication. Obtain an OAuth request token to be used for the rest of the authentication process. [...] Since this method is on behalf of an unauthenticated user, no access token or secret should be involved when signing or sending the request.*

These informal constraints require that no secret should be sent to the service when the authentication is initiated. Then all communication needs to use encryption provided by the SSL protocol. Although both versions of the standard authentication protocol are supported, the DropBox documentation recommends developers to use the latest version.

[2]Informal constraint taken ad litteram from https://viaforensics.com/resources/reports/best-practices-ios-android-secure-mobile-development/copy-paste/

[3]Text snippets from the Dropbox documentation https://www.dropbox.com/developers/core/docs

Informal constraints in standard security protocols

Dropbox documentation refers further to the standard description of the authentication protocol. The architects may use such documentation to gather additional constraints from the security guidelines informally described. For example, to prevent client impersonation, the security guidelines of the OAuth 2.0 protocol requires the client credentials, which include the password and the authentication token, to be kept secret[4].

> *A malicious client can impersonate another client and obtain access to protected resources if the impersonated client fails to, or is unable to, keep its client credentials confidential.*

Informal constraints in version control repositories

Informal security constraints are also available in bug reports. The security-related bugs are only reported to a private email to protect the users until developers fix the bug. Details on security constraint may still be available as comments in the version control where the security bug is fixed. UPMA is actively maintained and has an available repository that contains the following comment[5]:

> *Hide accnt details in screenshots and task manager. Security precaution to ensure a third party can't view account details left open when UPM was last used.*

The UPMA developer changed the code to ensure that another application cannot steal the password by taking a picture of the screen when the password is displayed in clear text. The developer changed the class `ViewAccountDetails` and set a security flag in the method `onCreate` (Fig. 6.6).

[4]Security guideline taken ad litteram from the authorization protocol http://tools.ietf.org/html/rfc6749#section-10

[5]A commit message in UPMA https://github.com/adrian/upm-android/commit/bd53100

```
GIT commit @bd53100

src/com/u17od/upm/ViewAccountDetails.java public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  +if (Utilities.VERSION.SDK_INT >= Utilities.VERSION_CODES.HONEYCOMB) {
  +  getWindow().setFlags(LayoutParams.FLAG_SECURE, LayoutParams.FLAG_SECURE);
  +}
  setContentView(R.layout.view_account_details);
```

Figure 6.6: Changes in UPMA 1.14 to prevent screenshots being taken by malicious apps when the password in clear text is displayed.

$$insecureIntents(g : SecGraph) \tag{6.57}$$

$$insecureIntents := g.getFlowToSink(\{IsConfidential.\texttt{True}, \tag{6.58}$$

$$IsEncrypted.\texttt{False}\}, \{TrustLevel.\texttt{Untrusted}\}) \tag{6.59}$$

$$isIntent := InstanceOf(\texttt{Intent}) \tag{6.60}$$

$$setObjectsProperty(TrustLevel.\texttt{Untrusted}, isIntent) \tag{6.61}$$

$$isPwd := IsInDomain(\texttt{AccountInformation}, \texttt{PWD}, \texttt{String}) \tag{6.62}$$

$$setObjectsProperty(\{IsConfidential.\texttt{True}, IsEncrypted.\texttt{False}\}, isPwd) \tag{6.63}$$

$$\{e1, e2, e3, e4\} \subseteq insecureIntents(G) \tag{6.64}$$

$$e1 := \langle \texttt{vad:VADetails}, \texttt{intnt:Intent}, \texttt{vad:VADetails} \rangle \tag{6.65}$$

$$e2 := \langle \texttt{vad:VADetails}, \texttt{s:String}, \texttt{pwd:String} \rangle \tag{6.66}$$

$$e3 := \langle \texttt{al:AccountsList}, \texttt{s:String}, \texttt{pwd:String} \rangle \tag{6.67}$$

$$e4 := \langle \texttt{aea:AddEditAccount}, \texttt{s:String}, \texttt{pwd:String} \rangle \tag{6.68}$$

Figure 6.7: A constraint that checks if the password is sent to an object of type `Intent`.

6.2.2 Information Disclosure in UPMA

This case study evaluates if Scoria can find an architectural flaw that violates the informal security constraints previously described. Confidential information, such as the password in clear text, should not be disclosed to objects of type `Intent` or `ClipboardManager`. In Scoria, the machine-checkable constraint *insecureIntents* is executed on the *SecGraph* extracted from annotated code of UPMA.

In this example, finding the untrusted sink is straightforward based on the type of the object. Finding what objects are confidential requires a brief inspection of the code to locate where the password concept is implemented. The inspection revealed that the class `AccountInformation` has the fields `username` and `password` to store confidential information in plain text.

$$insecureClipboard(g : SecGraph, isPwd : Condition, isClipboard : Condition) \quad (6.69)$$

$$insecureClipboard := \bigcup\{e\} \text{ s.t. } e : DataFlowEdge \in g.edges() \wedge \quad (6.70)$$

$$o \in g.getObjectsByCond(isPwd) \wedge crEdge : CreationEdge \in g.getEdgesByFlow(o, \texttt{Creation}) \text{ where} \quad (6.71)$$

$$e \in g.getEdgesByFlow(crEdge.dst, \texttt{DataFlow}) \wedge e.dst \in g.getObjectsByCond(isClipboard) \quad (6.72)$$

$$isClipboard := InstanceOf(\texttt{ClipboardManager}) \quad (6.73)$$

$$isPwd := IsInDomain(\texttt{AccountInformation}, \texttt{PWD}, \texttt{String}) \quad (6.74)$$

$$\{e1, e2\} \subseteq insecureClipboard(G, isPwd, isClipboard) \quad (6.75)$$

$$e1 := \langle \texttt{sr:SearchResults}, \texttt{cm:ClipboardManager}, \texttt{s:String} \rangle \quad (6.76)$$

$$e2 := \langle \texttt{accts:FullAccountList}, \texttt{cm:ClipboardManager}, \texttt{s:String} \rangle \quad (6.77)$$

Figure 6.8: The password is sent indirectly through creation edges to an object of type `ClipboardManager`

The constraint *insecureIntents* (line 6.57) uses object hierarchy to distinguish between different objects of type `String`, where only the password and account represent confidential information (line 6.62). A potential information disclosure exists for those objects of type `Intent` that receive such information. The constraint uses the *getFlowToSink* query (lines 6.58), which in turn uses indirect communication through object hierarchy and object reachability to check that a dataflow object contains or can reach into confidential information.

To check if the password is exposed to the clipboard, the machine checkable constraint *insecureClipboard* uses object reachability through creation edges (Fig. 6.8). The constraint checks if a dataflow edge with an untrusted sink refers to the destination of a creation edge, which in turn refers to a confidential object. Scoria finds a vulnerability in the class `AccountsList` that `FullAccountsList` and `SearchResults` extend where the unencrypted password is disclosed (Fig. 6.9). Since the clipboard is accessible to any other app, passwords stored by UPMA are vulnerable to eavesdropping when the user copies them.

6.2.3 Lessons Learned

Fig. 6.10 shows a fragment of the UPMA OGraph, elides the substructure of some objects, and included only the edges that provide support for the following observations.

```
1  class FullAcountList extends AccountsList{ ... }
2  class SearchResults extends AccountsList{ ... }
3  class AccountsList {
4    boolean onContextItemSelected(MenuItem item) {
5    View targetView = ...;
6     ...
7     // a copy of the password in cleartext is sent to clipboard
8     setClipboardText(getPassword(getAccount(targetView)));
9    }
10   void setClipboardText(String text) {
11     ClipboardManager clipboardManager = ...;
12     clipboardManager.setText(text);
13   }
14   String getPassword(AccountInformation account) {
15     return new String(account.getPassword());
16   }
17 }
18 class AccountInformation{
19   String username;
20   String password;
21   ...
22   String getPassword() {
23     return password;
24   }
25 }
26 class ViewAccountDetails {
27   void populateView(){
28     ...
29     TextView accountPasswordTextVeiw = ...
30     //IT's OK, no app can take a screenshot when the password is displayed
31     accountPasswordTextView.setText(new String(account.getPassword()));
32   }
33 }
```

Figure 6.9: Code where the password in clear text is disclosed to clipboard

Constraints may return false positives

An object graph merges multiple objects of the same type in one domain, which may lead to false positive edges due to a method invocation in a super class [87]. None of the dataflow edges highlighted by the constraints in the CERT examples is a false positive. However, one of four dataflow edges highlighted by *insecureIntents* in UPMA is a false positive due to excessive merging of objects of type `String` in the domain `SHARED`. A points-to edge exists from `intnt:Intent` to `s:String` in `SHARED`. A creation edge from `aea:AddEditAccount` to `s:String` refers to the confidential object `pwd:String` (line 6.68). The object `s:String`

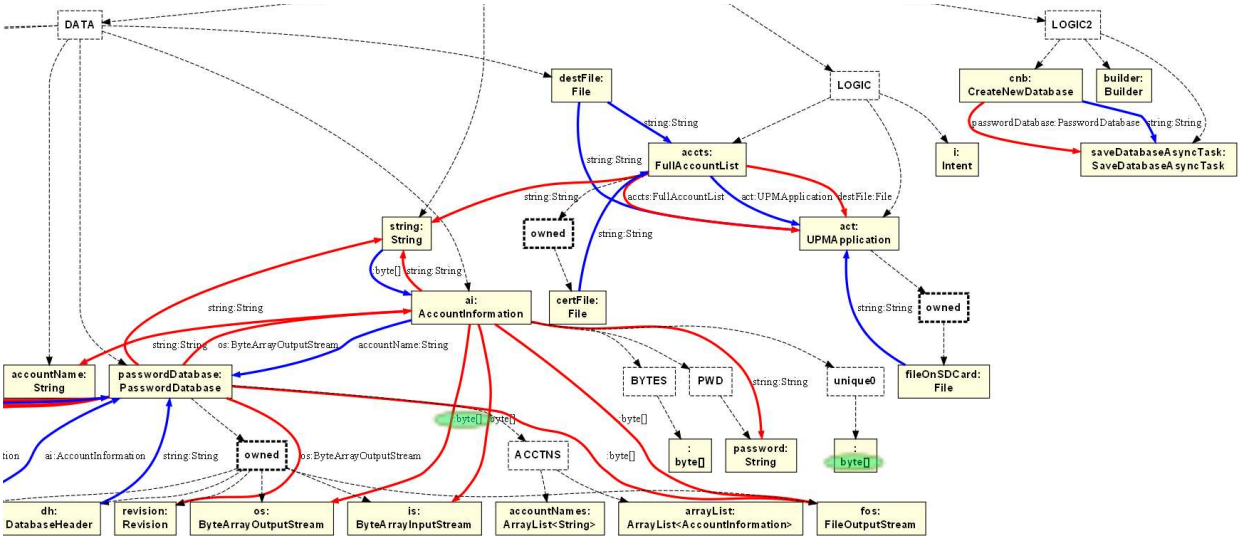


Figure 6.10: A fragment of the extracted UPMA *SecGraph* shows dataflow objects of type `File`, and a flow object of type `byte[]` from `ai:AccountInformation` to `os:ByteArrayOutputStream`

excessively merges objects of type `String` including the object of type `String` created when the UPMA user displays an account. Sending the password in plain text to be displayed is part of the intended UPMA functionality and is not a vulnerability because the UPMA developer prevented other applications to take a snapshot (Fig 6.6).

The analysis extracts dataflow edges that refer to objects and distinguishes between objects of the same type

The extracted *SecGraph* shows distinct objects of type `File`, where `dest:File` represents the file that stores the encrypted database, while `cert:File` represents a file that stores the encryption keys. The analysis shows a dataflow edge from `act:UPMApplication` to `accts:FullAccountList` that refer to `dest:File`. This edge is due to a feature in UPMA that allows the user to copy the database file to the external memory of the phone. If the dataflow edge were to show only the type of the object, it would be misleading because such a dataflow edge would indicate that the file containing the encryption keys is also copied and the encryption keys are exposed, which is not the case.

ScoriaX extracts several flow objects

While both objects of type `File` are nodes in the *SecGraph*, the analysis also extracts several flow objects. For example, a flow object `b:byte[]` is referred from the dataflow edge from `ai:AccountInformation` to `os:ByteArrayOutputStream`. This flow object is created during encryption when the object `pd:PasswordDatabase` translates `password:String` of `ai:AccountInformation` from plain text to a binary representation which is also confidential. The architects need to ensure that the binary representation only flows to the encryption service.

More precise ownership types produce more precise results

Developers can control the precision of the extracted graphs by placing objects in different domains. For example, the class `SaveDatabaseAsynkTask` has a field of type `Activity`. The field is the receiver of a method invocation `activity.getString()` which returns the flow object. The field is declared in `owner` domain. If all the instances of subclasses that extend `Activity` were in `LOGIC`, the analysis would show dataflow edges to the objects `act:UPMApplication` and `accts:FullAccountList`, which are of types that extend `Activity`. However, such edges would be false positives. One manual workaround is to have the developer define a separate domain, `LOGIC2`, and place these object in that domain, so the analysis does distinguish between these objects and `cnb:CreateNewDatabase`, thus avoiding these false positives (Fig. 6.10).

6.3 Evaluation on Benchmark

The section describes the evaluation based on an existing benchmark designed by related work [48]. Scoria is thus compared in terms of true positives, false positives, and false negatives with another research tool, FlowDroid [48] described in detail in Section 7.2.3 (page 159), and two commercial tools commonly used in industry Fortify [62] and App-

Scan [66] described in detail in Section 7.2.5 (page 161). This comparison is instructive because these tools use hard-coded constraints that check if confidential data flows from a source to an untrusted sink, where the list of sources and sinks can be automatically generated [20]. In contrast, Scoria combines annotations, a static analysis to extract a high-level representation, and constraints that are separated from extraction.

For evaluation, I include hand-selected test cases from DroidBench [48] because is one of a few studies that have the tools and the dataset publicly available to allow direct comparison. I excluded test cases from the same equivalence class that target the same behavior of a static analysis and tests that focus on code constructs specific to Java such as anonymous classes or static initialization blocks (Table 6.2). The benchmark is divided into equivalence classes that target the specifics of static analyses such as callbacks, collections, object-sensitivity, inter-application communication, and lifecycle of objects created by the Android framework. Other approaches either are evaluated on commercial systems such as popular Android apps, or are evaluated only on a few test cases, which do not allow a systematic comparison. Popular apps restrict our evaluation to static analyses that use bytecode because their source code is often unavailable. Scoria requires the source code to be available and can be used to find vulnerabilities proactively before the release.

The evaluation uncovered some classes of vulnerabilities that DroidBench lacked, so the original benchmark was extended with new test cases, which are related to authentication, encryption, and injection attacks. Some of the new test cases were inspired the examples in the CERT Oracle Secure Coding Standard for Java [81]. Then, I proposed test cases related to bypassing authentication, exploitable service and violation of least privilege principle. Since the test cases I contributed to the benchmark may be useful for other researchers, I am making them publicly available [3].

The Scoria evaluators who were previously unfamiliar with Scoria or FlowDroid received instructional training on adding annotations, running the static analysis, and writing constraints. The test cases IntentSink1 and ObjectSensitivity1 from DroidBench were used in

Table 6.2: Selected test cases from DroidBench

	Selected		DroidBench	
Equivalence class	**tests**	**vulns.**	**tests**	**vulns.**
Arrays and Lists	1	0	3	0
Callbacks	5	9	6	10
Field and Object Sensitivity	6	2	7	2
Inter-App Communication	3	3	3	3
Lifecycle	5	5	6	6
General Java	3	2	5	4
Android-Specific	5	3	5	3
Implicit Flows	4	8	4	8
TOTAL	32	32	39	36

Table 6.3: Recall and precision based on DroidBench

	AppScan	**Fortify**	**FlowDroid**	**Scoria**
TP, higher is better	13	14	31	35
FP, lower is better	4	3	6	6
FN, lower is better	19	18	12	8
Precision	76%	82%	84%	85%
Recall	41%	44%	72%	81%
F-measure	0.53	0.57	0.77	0.83

the training. The evaluators then completed the other test cases mostly on their own, but got frequent feedback from the Scoria designers. The evaluator did not rerun FlowDroid on the test cases that were reused from the original DroidBench benchmark, but did run FlowDroid on the additional test cases.

6.3.1 Quantitative Results

Table 6.4 presents the results of the evaluation and its rows show the selected test cases grouped in equivalence classes, where the last rows has test cases in the extended benchmark. The first column with the name of the test cases is followed by several columns that emphasize which features of Scoria the evaluators used. The last four groups of columns present the results for each tool showing the number of true positives (TP), false positives (FP), and false negatives (FN).

This section reuses the existing results from the related study [48] without rerunning

Table 6.4: Results of comparing approaches that find security vulnerabilities. Results on AppScan, Fortify and FlowDroid are reused from the original study [48].

	provenance (a)	transitivity (b)	hierarchy (c)	reachability (d)	traceability (e)	indirect communication (f)	object security properties (g)	edge security properties (g)	typechecker warnings	extraction warnings	AppScan TP	AppScan FP	AppScan FN	Fortify TP	Fortify FP	Fortify FN	FlowDroid TP	FlowDroid FP	FlowDroid FN	Scoria TP	Scoria FP	Scoria FN
Arrays and Lists																						
ListAccess1		✓					0	0	0	3	0	1	0	0	1	0	0	1	0	0	1	0
Callbacks																						
Button1							2	0	1	6	0	0	1	1	0	0	1	0	0	1	0	0
Button2							2	0	1	8	1	0	2	1	0	2	3	1	0	3	0	0
LocationLeak1							2	0	6	4	0	0	2	0	0	2	2	0	0	2	0	0
LocationLeak2							2	0	6	2	0	0	2	0	0	2	2	0	0	2	0	0
MethodOverride1		✓					0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Field and Object Sensitivity																						
FieldSensitivity1		✓					0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
FieldSensitivity3		✓					0	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0
FieldSensitivity4		✓					0	0	1	3	0	1	0	0	0	0	0	0	0	0	1	0
InheritedObjects1							2	0	1	3	1	0	0	1	0	0	1	0	0	1	0	0
ObjectSensitivity1		✓					0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
ObjectSensitivity2		✓					2	0	2	4	0	1	0	0	0	0	0	0	0	0	2	0
Inter-App Communication																						
IntentSink1				✓	✓		2	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0
IntentSink2							2	0	0	6	1	0	0	1	0	0	1	0	0	1	0	0
ActivityComm1	✓						0	0	1	4	1	0	0	1	0	0	1	0	0	1	0	0
Lifecycle																						
BReceiverLifecycle1							2	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle1		✓			✓		0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle2							2	0	1	4	0	0	1	1	0	0	1	0	0	1	0	0
ActivityLifecycle3							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0
ServiceLifecycle1							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0
General Java																						
Loop1							2	0	2	3	1	0	0	0	0	1	1	0	0	1	0	0
SourceCodeSpecific1		✓					0	0	2	3	1	0	0	1	0	0	1	0	0	1	0	0
UnreachableCode							2	0	1	1	0	0	0	0	1	0	0	0	0	0	1	0
Miscellaneous Android-Specific																						
PrivateDataLeak1		✓					0	0	3	7	0	0	1	0	0	1	1	0	0	1	0	0
PrivateDataLeak2							2	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0
DirectLeak1							2	0	0	4	1	0	0	1	0	0	1	0	0	1	0	0
InactiveActivity							2	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0
LogNoLeak							2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Implicit Flows																						
ImplicitFlow1							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow2							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow3							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow4							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
Extended Benchmark																						
ACipher	✓						0	0	0	1							0	0	0	0	0	0
ACipher2	✓						0	0	0	0							0	1	0	0	0	0
AToken1			✓	✓		✓	2	0	0	6							3	0	0	3	0	0
AToken2			✓	✓		✓	2	1	0	6							1	2	0	1	0	0
ASocket			✓			✓	2	0	2	5							1	1	0	1	0	0
AActivity				✓			2	0	0	3							0	0	1	1	0	0
ARuntime					✓		1	0	0	6							1	0	0	1	0	0
DataContainer				✓			2	0	2	4							2	0	0	2	0	0
AChat	✓						0	0	0	2							0	1	1	1	0	0
SecretViewer	✓						1	0	3	1							0	0	1	1	0	0

FlowDroid, AppScan and Fortify on the original benchmark. Since FlowDroid outperforms the commercial tools in terms of both precision and recall based on the original benchmark, the corresponding cells in the extended benchmark are empty.

Compared to FlowDroid, Scoria has higher recall, precision and a similar F-measure[6] (Table 6.3). Based on the original test cases from DroidBench, Scoria finds one vulnerability and avoids one false positive of FlowDroid, but reports five additional false positives. The extended benchmark highlights cases where Scoria can do better than FlowDroid, which misses three vulnerabilities that Scoria finds and also reports five false positives.

6.3.2 Qualitative Analysis

The results of the evaluation show that by using object provenance and indirect communication constraints, Scoria can find security vulnerabilities such as information disclosure and tampering. Comparing to other research approaches, Scoria performs better in terms of recall and precision due to the flexibility Scoria provides for fine-tuning the annotations that provide precision about aliasing, for writing constraints, and for assigning security properties. Scoria also performs better in terms of precision and recall than two commercial security tools when targeting the same types of security vulnerabilities. The evaluation also highlights several limitations of Scoria in terms of precision due to ScoriaX being flow-insensitive, and in terms of scalability due to the manual process of adding annotations.

Scoria meets most of the requirements from Section 1.4 (page 9). The rest of the section revisits the requirements and presents in more detail several lessons learned based on Scoria evaluation.

Unsound call-graph reduces recall

The results for the commercial approaches indicate a lower recall for the Callbacks and Lifecycle equivalence classes of DroidBench because of a precomputed call graph. Extract-

[6]The weighted harmonic mean of precision and recall

ing a call graph is challenging in the presence of callbacks that are methods invoked by a framework in an order predefined by the object lifecycle. Security vulnerabilities exist for example when one callback stores confidential data into a field, and another callback reads the value of the field and discloses the confidential data into an untrusted sink. Callbacks often implements the Observer design pattern, which uses subjects and observers and notifies the registered observers when the state of a subject changes.

FlowDroid is optimized to find vulnerabilities in applications built on the Android framework. It models the lifecycle of an object of type `Activity` by adding the callbacks as entry points in the call graph and assumes that all the callbacks are invoked. Therefore, FlowDroid has a better recall compared to AppScan and Fortify for the LifeCycle test cases, LocationLeak1 and LocationLeak2. FlowDroid reports, however, a false negative for SecretViewer, where information disclosure occurs when a confidential document is changed and a notification is sent to all the objects of a type that implement the Listener interface. FlowDroid uses an unsound call graph and cannot resolve the type behind the Listener interface; therefore, it misses the vulnerability.

To achieve high recall, ScoriaX safely assumes that every method declared on an instantiated class may be invoked, and avoids the use of an unsound call graph. This assumption reduces the computational complexity and increases recall for the aforementioned test cases. Admittedly, the assumption is a source of imprecision of Scoria as the UnreachableCode test case highlights.

By filtering, approaches gain precision, but reduce recall

Because one vulnerability can compromise the whole system, Scoria avoids sacrificing recall for precision. This is in contrast to FlowDroid, which uses the value of attributes in configuration files to filter the call graph and ICFG to gain more precision. In Android, for each object of type `Activity`, a configuration file (`manifest.xml`) contains values for attributes that define the behavior of the object at runtime. For example, if a class that

extends `Activity` has the attribute `android:enabled` assigned `false`, the framework cannot instantiate any object of such a class. FlowDroid filters the fragment of the ICFG that is related to such a class. By filtering, FlowDroid avoids a false positive in InactiveActivity, but fails to find the vulnerability in AActivity, where the attribute is changed at runtime bypassing the configuration file. Scoria is more conservative and assumes that all classes may be instantiated at runtime. Scoria finds the vulnerability in AActivity and returns one false positive for InactiveActivity.

Some classes of security vulnerabilities are hard to find

Security vulnerabilities that are hard to find seem to involve implicit flows and immutability; I discuss each one in turn.

An implicit flow is a dependency between the predicate of a conditional statement and the expression executed based on the value of the predicate. For example, if an application allows unlimited login attempts and returns a binary response, a malicious user can use brute-force attack based on a dictionary to guess the passwords although there is no explicit flow of the confidential password to an untrusted sink. None of evaluated approaches supports implicit flows[7] (Table 6.4).

Another class of vulnerabilities that are hard to find involves immutability. Consider for example an object of type `HashMap` that has values representing credit card information and keys representing the address of the credit card owner. If the state of the object used as a key is changed at runtime (i.e., the object is mutable) the credit card information can no longer be looked by its previous key, which may result in a denial-of-service attack. To overcome this difficulty, researchers proposed approaches for enforcing immutability [105] and determining *pure* methods that are guaranteed to not change the state of the receiver object [109]. Scoria does not consider immutability.

[7]Milanova et al. proposed a static analysis that extracts implicit flows edges in the ICFG [77] and FlowDroid reports support of implicit flows as work in progress [26].

Summarization increases recall

Summarizing multiple runtime objects that have the same conceptual purpose with one representative (*H1f*) increases recall when the objects are created by different object allocation expressions in the code. Finding such vulnerabilities would otherwise require an analysis that keeps track values that flow through library code (which may not be available), or through files (which may be hard to track). The test case Loop1 obfuscates an object of type `String` that represent the device identification number by adding an intermediate character between any two characters in the initial object. The obfuscated object still represents confidential data; hence, the architect decides to use one representative for both objects by placing them in the same domain. A runtime dataflow with an object of type `SmsManager` as a destination would only refer to the obfuscated object. In Scoria, the architect assigns *IsConfidential.True* to the representative of both objects, and the constraint reports a true positive. FlowDroid and AppScan keep track of the transformations by propagating the security property from the argument of the method `concat` of the class `String` to the return value and report the vulnerability, while Fortify fails to report it.

Soundness does not generate an excessive number of false positives

Although for some test cases FlowDroid is more precise being flow-sensitive, Scoria compensates for flow-insensitivity in other test cases using domain-sensitivity. ScoriaX is a sound (*H1c*) flow-insensitive, but domain-sensitive analysis (*H1b*). The precision at the level of objects and edges is reflected in finding vulnerabilities. On average, 80% of vulnerabilities reported by Scoria are true positives, thus Scoria achieves a high precision.

Scoria reports more false positives than FlowDroid for the test cases FieldSensitivity4 and ObjectSensitivity2, where a confidential and a non-confidential value are assigned to the same variable. Since ScoriaX is flow-insensitive and the domain of an abstract object does not change at runtime, Scoria assumes the worst case, i.e., that the variable may alias a confidential object. For other test cases, Scoria compensates by distinguishing between

objects of the same type in different domains. Then, Scoria is more precise and avoids the false positive reported by FlowDroid. For Button2, a variable `imei` is assigned `null` before is concatenated to a constant. In Scoria, the architect inspects the code and fine-tunes the annotations such that the result of concatenation is in `SHARED` while the object `imei:String` is in the domain `DATA`.

Separating constraints from extraction increases recall

Scoria is semi-automated (**SC2**) by allowing the architects to write machine-checkable constraints. It separates constraints from extraction (**SC3**) and is able to find a vulnerability that FlowDroid misses. FlowDroid uses a constraint that is only based on transitive communication of objects. Keeping track of all assignments is challenging because some occur in the code frameworks and libraries and this code may not always be available. For the test IntentSink1, the confidential information is first passed to an object of type `Intent`, which is then passed as an argument to an untrusted sink through some variables in the framework. FlowDroid fails to find these vulnerabilities since it does not analyze the framework. By using constraints based on object traceability and object reachability, Scoria reports the vulnerability. The constraint checks if the same information that leaves the source is passed to the sink through some other objects. Thank to the domain annotations, it is easier for Scoria to keep track of information flow because it merges objects of the same type in the same domain.

Object provenance increases recall

Scoria supports a constraint in terms of object provenance, which is used to find the vulnerability in AChat, where the same object that flows from the application to the service should not flow from an external client to the service. The service of AChat allows a registered user to search for existing phone numbers in a database. A malicious client can use the service to register a fake user that searches for all possible phone numbers in a region that

are valid and registered by the service. Such an attack circumvents the use of encryption and exposes confidential information (valid phone numbers in a region) although the service is not explicitly disclosing phone numbers[8]. To find such a vulnerability, architects need flexibility to adapt and write application-specific constraints, which the designer of a security approach may not consider.

Object provenance and indirect communication increase precision

Scoria allows architects to write constraints in terms of object provenance (*H2a*) and indirect communication (*H2f*), which increase the precision. Scoria can implement the constraints of FlowDroid using object transitivity and object reachability through only points-to edges. Unsurprisingly, for most of the test cases in DroidBench, using such a constraint provides sufficient precision (Table 6.4, column 3). Scoria has other features that allow architects to increase precision. For example, Scoria considers reachability through other types of edges such as creation edges in addition to the points-to edges. For the test cases in the extended benchmark, avoiding false positives requires using object provenance, edge traceability, and object hierarchy in addition to object reachability. For ASocket, the architects writes a constraint that uses indirect communication and checks that no confidential object flows to a descendant of an object of type `Socket`, which does not provide encryption. Using object hierarchy, the constraint distinguishes between two objects of the same type, `SocketOutputStream`. The first object is a descendant of an object of type `Socket` and the second is a descendant of an object of type `SSLSocket`. Thus, Scoria avoids a false positive that FlowDroid reports if it considers the method `SocketOutputStream.write` an untrusted sink.

[8]This test case is inspired from the SnapChat attack [51]

Application-specific security properties increase precision

Scoria separates security properties from extraction (**SC3**), and allows the architects to assign security properties to objects and edges on the extracted object graph. The same security properties can be reused across applications for classes in the Java library and frameworks. In several test cases however, assigning security properties requires application-specific knowledge. In FlowDroid, this requires using sources and sinks that are application-specific methods, while in Scoria it requires knowledge about locally declared domains or types that architects can gain by querying the OGraph. Assigning application-specific properties increases the precision for several cases, including the precision of FlowDroid in the test case ACipher where the architect adds to the list of sources the method declaration `decrypt` of the class `Service`. Otherwise, if the architect were to use as a source the method `doFinal` of the class `Cipher` that can encrypt or decrypt data, FlowDroid would return a false positive. This also occurs in test case ACipher2, where two objects of type `Service` exist, and the class `Service` has only one method `run` that decrypts or encrypts data. For ACipher2, FlowDroid reports a false positive, which is avoided by Scoria using object provenance.

Assigning security properties on edges can also increase precision. For AToken2, a client object has a field marked as transient, and this field is not serialized. The field represents confidential information, but information disclosure does not exist even if the client object is serialized into an untrusted file. To avoid the false positive, Scoria uses the *IsTransient* property for the points-to edge. This property is independent of the serialization library being used. By assigning security properties only to objects, FlowDroid reports two false positives.

Assigning properties to objects instead of declarations increases precision

Scoria uses a runtime architecture (**SC1**) and distinguishes between objects of the same type that have different security properties. Section 6.1 discusses why the use of a class declaration to assign security properties can lead to false positives. By using method decla-

rations to assign security properties, FlowDroid is less precise than Scoria for test cases where the same method is used in different contexts. Consider for example the method `doFinal` of `Cipher`, which returns encrypted or decrypted data depending on the context, and the method `write` of the class `SocketOutputStream`, which does not disclose confidential data if the parent of the object of type `SocketOutputStream` is of type `SSLSocket`.

6.4 Threats to Validity

The test cases in the extended benchmark are small, consisting of fewer than 100 lines of non-comment, non-blank lines of code, with at most 5 classes. Despite being small, the micro-benchmark allows us to understand the differences between Scoria and other approaches in terms of precision and recall. The test cases are realistic and cover a wide range of vulnerabilities that may occurred in practice.

In the evaluation using the benchmark (Section 6.3), the expertise and experience of the evaluators using the approaches may influence the results. This issue is more pronounced with Scoria since it is not a push-button approach and involves human judgment in adding annotations, writing constraints and interpreting the results. In this case, the evaluators had little prior experience. In the presence of false positives, the architects can refine the constraints to use object hierarchy, object reachability, and the traceability to code that the security model provides. Some of these features of Scoria are used more than others, which might happened because some of the advanced features of Scoria are more difficult to use in practice. It would be interesting to investigate how much effort it takes to refine the constraints, and which features require more effort.

Another threat to validity is that the evaluators have access to documentation in each test case that describe what constitute confidential data or untrusted sink, along with the number of existing vulnerabilities. In practice, and for the UPMA case study (Section 6.2), such information is not available and the process stops based on the available resources.

The Scoria evaluators first minimized the number of extraction and annotation warnings. In the second phase, they focused on writing constraints to minimize the number of false positives and false negatives, and rarely changed the annotations. By running again the typechecker, the evaluators ensured that the annotations and the code are consistent. It would be interesting to evaluate different exit criteria such as measuring the quality of the annotations [134].

Another threat to validity is related to the visualization of the extracted graph. For the DroidBench test cases, the abstract object graphs are small and the evaluators relied on exported object graph as pictures while writing constraints. For larger systems, they would need an interactive visualization tool to collapse and expand the visualization of objects and lift edges to keep the graphs manageable and reduce effort. In this thesis, I assume that a hierarchical OGraph, can be visualized using nested boxes (see details in [4, Sec. 2.4.3]). The visualization can use different abstraction mechanisms such as abstraction by types within a domain [4, Sec. 3.4.2], and achieves 1x–10x reduction in the number of abstract objects displayed [134]. This kind of visualization enable high-level understanding, while the details are still available as needed by expanding the substructure of a displayed object. The examples used in DroidBench are small enough that it is possible, as well as useful and instructive to look at the OGraph. For example, evaluators can see where the analysis loses precision for the value-flow analysis, or where additional precision is needed.

Scoria is focused on finding architectural flaws related to connectors in a runtime architecture and focuses only on the runtime structure. Architectural flaws may also be related to the state of the object, and the abstract object graph does not capture such information. Both static [13] and dynamic analyses [137] were proposed to keep track of changes in the states of objects.

6.5 Summary

This chapter provides quantitative and qualitative support to show that Scoria meets the security analysis requirements. Scoria is a semi-automated approach (**SC2**) that supports ARA by using the *SecGraph* as an approximation of the runtime architecture (**SC1**). The architects can write informal security policies as machine-checkable constraints that are executed against the *SecGraph*. The constraints are extensible (**SC3**) and can express CERT rules and security policies from existing documentation. By using constraints in terms of object provenance and indirect communication, Scoria achieves more than 80% recall and 80% precision on finding information disclosure and tampering. The results are better than the ones of commercial approaches, and for some test cases, Scoria can find vulnerabilities that information-flow based approaches such as FlowDroid [48] miss. The next chapter (Chapter 7) discusses related work in detail.

Credits

I would like to thank Ebrahim Khalaj, Sumukhi Chandrashekar, and Dajun Lu who annotated most of the DroidBench test cases and wrote constraints. Ebrahim also contributed to the test cases in the extended benchmark and ran FlowDroid to provide the numbers in Table 6.4.

Acknowledgements

I would like to thank the EC-SPRIDE group for making FlowDroid and DroidBench available. We are also making available our extended benchmark [3].

Chapter 7 Related Work

This chapter highlights conceptual differences between approaches that use a program analysis to find security vulnerabilities in object-oriented programs. The approaches focus on web applications and mobile applications where one security vulnerability can affect many users. Scoria consists of a static analysis that extracts a sound, hierarchical object graph where the architects can fine-tune the precision of the extraction analysis using ownership types annotations. To find information disclosure and tampering, Scoria allows architects to write constraints and assign security properties to abstract objects and edges, where the constraints and security properties are separated from the extraction analysis.

Other than Scoria, several approaches target the same problem of finding information disclosure and tampering using a static analysis. I focus on general approaches without targeting applications implemented in a specific object-oriented framework (Table 7.1). The static taint analyses [21, 43, 49], information flow control and the tainting type system [37, 92] combine constraints and extraction. Although the constraints consider transitive dataflow communication and reachability, these approaches may miss some of the vulnerabilities that can be found by the separate constraints in architectural-level approaches [115, 16] and Program Query Language (PQL) [84]. Most approaches are flexible in allowing architects to assign security properties but at different levels of granularity such as objects, variables, expressions, method and class declarations.

The chapter has the following structure. Section 7.1 describes the static and dynamic analyses that extract object graphs focusing on how these object graphs can be used for reasoning about security. Section 7.2 describes existing approaches for finding security vulnerabilities and compares them with Scoria. Section 7.3 discusses how related approaches are evaluated.

Table 7.1: Comparison of static approaches for finding vulnerabilities

	Scoria	Static Taint Analysis	Information Flow Control	Tainting Type System	Program Query Language	Architectural
automation	semi-automated	automated	semi-automated	semi-automated	semi-automated	semi-automated
constraints	separate	combined (transitive comm. + reachability)	combined (transitive comm. + reachability)	combined (transitive comm.)	separate	separate
properties	abstract objects, abstract edges	method declarations	expressions	variable declarations	abstract objects	method/class declarations
program representation	hierarchical object graph	ICFG	SDG	AST	flat object graph	UML diagrams
soundness	proof	aim	proof	conjecture	conjecture	no
aliasing	ownership types	separate may-alias (SOOT, WALA)	separate may-alias (WALA)	type system	separate may-alias (BDD)	no
summarization	abstract objects	sources and sinks	sources and sinks	Type, property	abstract object	class
supports legacy code	yes + annotations	yes + configuration files	yes	yes + annotations	yes	yes
hierarchy	object-domain hierarchy	no	no	no	no	classes and packages
precision	domain-sensitive, object-insensitive, flow-insensitive	object-sensitive, flow-sensitive or flow-insensitive	object-sensitive, flow-sensitive	flow-insensitive	context-sensitive, flow-insensitive	level of classes
scalability limited by (larger system analyzed)	annotations (30KLOC)	aliasing (>1MLOC)	aliasing (100KLOC)	annotations (100KLOC)	aliasing (100KLOC)	extraction (>500KLOC)

7.1 Extracting Object Graphs

Section 2.1 (page 14) discusses how an object graph approximates a runtime architecture and what requirements an analysis must meet in order to extract an object graph that is suitable for ARA. This section describes static and dynamic analyses that extracts object graphs and discuss how these analyses meet the identified requirements in Section 1.5.1 (page 10).

7.1.1 Static Analyses

A static analysis works at compile time and uses as input the source code or compiled bytecode. To handle the complex expressions in the code, the static analyses often transform complex expressions into an intermediate representation as a three-address code [128]. Each three-address code expression has at most three operands and is typically a combination of an assignment and a binary operator or a method invocation that has only variables as arguments. This transformation reduces the complexity of a static analysis. To represent the dynamic structure of the program, static analyses extract object graphs where a node is an abstract object and edges represent relations between abstract objects.

Flat object graphs

Existing static analyses extract object graphs with points-to edges [87, 76, 114] or dataflow edges [116, 67]. Most of these graphs are non-hierarchical and mix objects representing implementation details with architecturally relevant objects. Moreover, the label of a dataflow edge is either missing or refers to a type rather than an abstract object. This detail becomes crucial for writing constraints in terms of object provenance. Some of the extracted object graphs [116, 67] are unsound and show multiple representatives for the same runtime object. Then, an architect can potentially assign different security property values for the same runtime object. Other static analyses extract sound object graphs, but with points-to edges only [96, Section 11.1.2].

For a given method invocation, analyses extract different sources and destinations for a dataflow edge. For a method invocation `b.m(c)` in the context of an object `a:A`, Spiegel's analysis [116] extracts an export dataflow edge from the context object `a:A` to the receiver `b:B`, and an import dataflow edge from `b:B` to `a:A`. On the other hand, Milanova's analysis [75, Figure 4.1] extracts an edge from the receiver object `b:B` to the argument object `c:C` any without labels. Similar to Spiegel's analysis, ScoriaX extracts dataflow edge refers to an

object. That is, the analysis creates an export dataflow edge from the context object `a:A` to the receiver object `b:B` and the edge refers to the argument `c:C`.

Jackson and Waingold [67] propose the Womble analysis that extracts a flat object graph with points-to edges. On the points-to relation, the edge label is the name of the field in the corresponding field declaration. Womble however does not address the aliasing challenge and extracts unsound object graphs.

Hierarchical object graphs

Other static analyses also use annotations to extract hierarchical object graphs. For example, Lam and Rinard [73] propose a static analysis to extract an object model by merging objects based on developer-specified annotations (tokens). The tokens provide precision such that the analysis handles aliasing and extracts models for "subsystem access", "call/return interaction", and "heap interaction", which are similar to the dataflow edges ScoriaX extracts. The label on the edges representing call/return interaction is a token that represents objects passed as arguments. If it were to follow the same idea, ScoriaX would show domains as labels. Domains are different from tokens because the set of tokens is fixed and statically declared such that the extracted object model has a predefined hierarchy depth.

7.1.2 Framework-Based Static Analyses

Existing program analyses have already been evaluated as clients of compilers [114]. Once the analyses became available as open-source framework, researchers other than the original authors of the analyses use them to find security vulnerabilities. For example, the static analysis frameworks such as SOOT [72] and WALA [65] have initially been used to compare different variants of point-to analyses. Recently, they are part of approaches that find security vulnerabilities in Android applications [21, 43, 49]. Framework-based static analyses extract graphs with nodes that represent variables, method declarations, and expressions. This section describes some of the graphs in detail.

System Dependency Graph (SDG)

SDG has nodes representing statements such as object allocation expressions, which represent abstract objects and method call-sites. An edge in SDG represents data and control flow between these expressions. A SDG approximates the information flow that occurs at runtime and can be extracted by static analyses frameworks such as WALA [65]. An SDG is used in Information Flow Control approaches discussed later in this chapter (Section 7.2.6). A SDG includes explicit data flows that are due to assignments and implicit flows. An implicit flow [77] exists when the execution of an expression is controlled by another expression, as it is the case for conditional statements.

Abstract System Dependency Graph (ASDG)

An Abstract System Dependency Graph (ASDG) [30] is derived from a SDG and it has nodes representing expressions has edges representing method invocations (control flow), and assignments (data flow). Since an SDG shows a node for every expression in the bytecode, the level of detail can be overwhelming for an architect to write constraints (Fig. 7.2). An ASDG is a closer representation of a code structure with nodes representing classes and methods, and is often used in program comprehension [103].

Interprocedural Control Flow Graph (ICFG)

Another static analysis framework, SOOT [72], extracts an ICFG, which is an equivalent of a SDG where the body of each method is represented once as a Control Flow Graph (CFG). Both SDG and ICFG are used in static taint analyses to find security vulnerabilities in Android applications [21, 43, 49] (Section 7.2.3).

Precision of a framework-based static analysis

The architects have a low tolerance for false positives. A precise static analysis can be for example flow-sensitive or context-sensitive. However, a precise static analysis that is flow-,

context-, and object-sensitive does not scale for a large number of variables [114]. Another static analysis framework (DOOP [113]) increases scalability by implementing the analysis in the declarative language Datalog [63]. Still, the parameterized object-sensitive analysis is implemented only for $k = 1$ and $k = 2$ with limited scalability [114]. ScoriaX uses a domain as a context, and is *domain-sensitive* and object- and flow-insensitive (Section 2.4, page 25). Instead of switching sensitivity flags for the whole program, architects can increase the precision as needed. They have the recourse of fine-tuning the annotations by preventing two objects from being merged by placing them in different domains.

Another limitation of these frameworks is that the may-alias analysis uses a precomputed call graph, which in turn uses the result of may-alias analysis. In the presence of callbacks and polymorphism, computing a sound call graph is challenging [14]. Callbacks often occur in the presence of the observer design patterns, when an event triggered by the subject invokes a method of the observer.

Consider for example the code fragment in Figure 7.1, where the method `update` of the class `Listener` is invoked in the method `notifyObservers` of the class `BaseChart`. Since `Listener` is an interface, and several classes that implement this interface exist, the static analysis needs to determine the actual method invoked at runtime. An unsound call graph shows only the method update of the interface `Listener`. An over-conservative sound approach assumes that the methods `update` of all the classes that implement `Listener` might be invoked. At runtime, only the method `update` of the classes `Model` is invoked because only the object `model:Model` is registered as an observer. However, the sound call graph shows that the method `update` of `BarChart` and `PieChart` are also invoked (Fig. 7.2(a)). By using annotations, ScoriaX is more precise and the OGraph shows no edges between `pieChart:PieChart` and `barChart:BarChart` (Fig. 7.1).

```
class Main<OWNER> {
  domain DOC, VIEW;
  BarChart<VIEW, DOC> barChart = new BarChart();
  PieChart<VIEW, DOC> pieChart = new PieChart();
  Model<DOC, VIEW> model = new Model();
  void run(){
   barChart.addListener(model);
   pieChart.addListener(model);
   barChart.notifyObservers(); //no dataflow
   pieChart.notifyObservers(); //no dataflow
  }
}
class Model<OWNER, V> extends Listener<OWNER> {}
class BarChart<OWNER,M> extends BaseChart<OWNER,M>{}
class PieChart<OWNER,M> extends BaseChart<OWNER,M>{}
interface Listener<OWNER> {
  public domain DATA; //public domain
  void update(Msg<lent> msg);
}
class List<OWNER, T<ELTS>> {//generic type T
  T<ELTS> value; // ELTS is a domain parameter
}
```

```
class BaseChart<OWNER,M>
extends Listener<OWNER> {
  domain OWNED; //domain declaration
  List<OWNED, Listener<M>> listeners;
  Msg<DATA> vTOm = new MsgVtoM();
  //DATA is a public domain
  public Msg<DATA> getMsg(){
    return mTOv;
  }
  void addListener(Listener<M> l) {
   listeners.value = l;
  }
  void notifyObservers() {
   Listener<M> l = listeners.value;
   //M maps to DOC
   //for l analysis lookup objects
   // of a subtype of Listener in DOC
   // and creates edges only to model
   // not to barchart or piechart
   l.update(vTOm);
  }
}
```

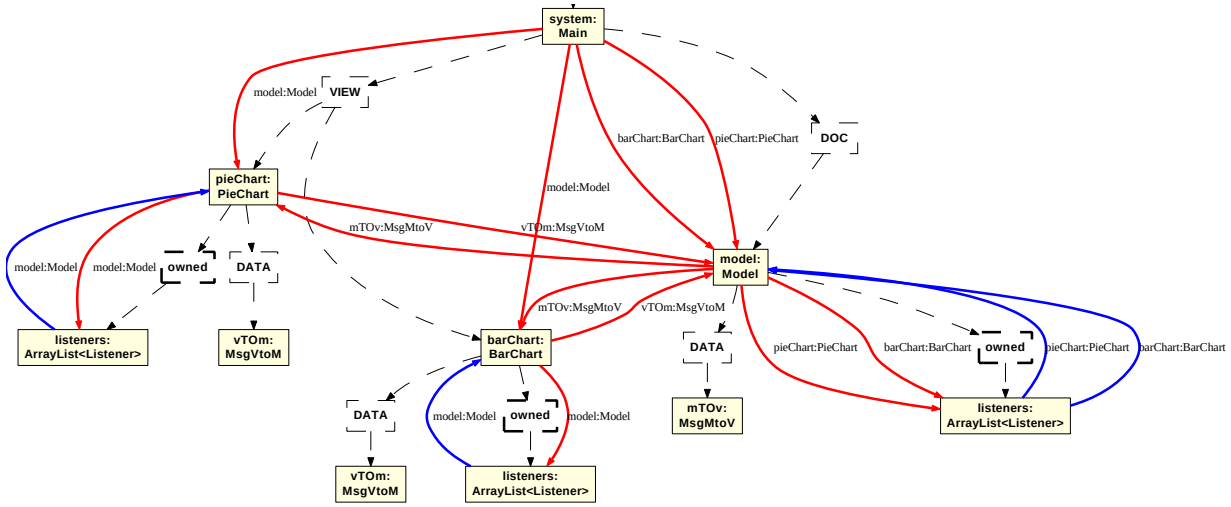
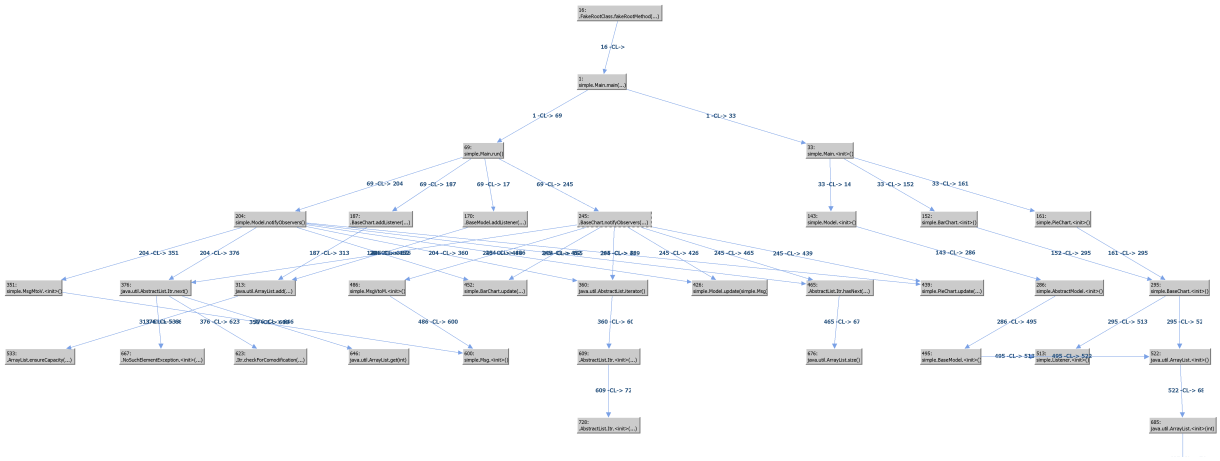


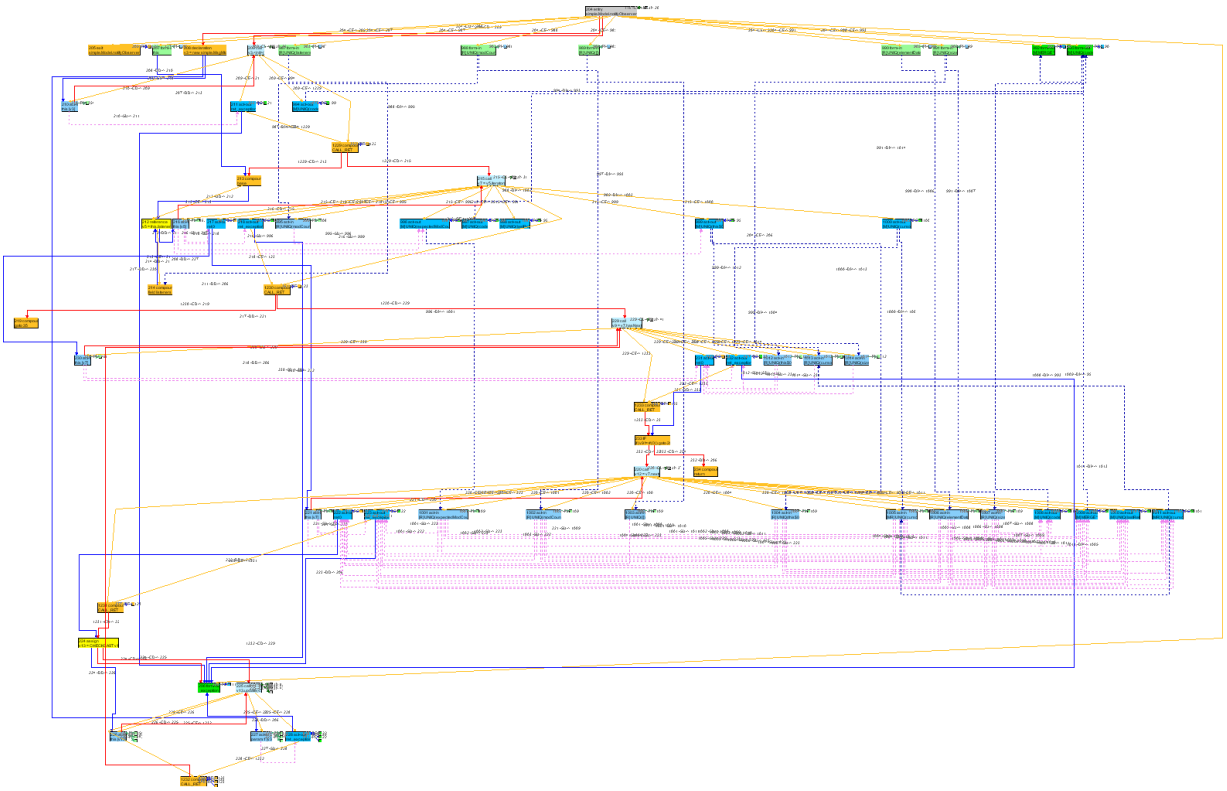
Figure 7.1: Listeners code fragments. The complete code is in [130].

7.1.3 Dynamic Analyses

A dynamic analysis can be more precise than a static analysis, and can extract a hierarchical object graph [74, 89, 83]. If the input of a dynamic analysis is a heap snapshot, the analysis can extract only points-to relations between objects, which are persistent relations at runtime. On the other hand, if the input is an execution trace, which has information about method invocations and their arguments in addition to the instantiated objects, a



(a) Sound call graph shows a false positive edge from `Model.notifyObservers()` to `Model.update(Msg)` and from `BaseChart.notifyObservers()` to `BarChart.update(Msg)`



(b) A fragment of the SDG obtained by expanding the invocation `notifyObservers` of the class `Model`.

Figure 7.2: A sound call graph and SDG for the Listeners example in Fig. 7.1 extracted using the static analysis JOANA [53]. Fragments of the SDG are available by expanding the nodes in the call graph. Nodes represent bytecode expressions, and edges represent value flow and control flow edges between variables and expressions. The labels on the edges show the identifiers of the source and destination. Zoom in to 400% for details.

dynamic analysis can extract dataflow edges. A dynamic object graph may have thousands of nodes and a dynamic analysis requires extensive graph summarization to obtain an abstracted graph [89, 56]. Storing and recording execution traces can be problematic due to their size. Dynamic analyses often use filters to keep the size of execution traces manageable. However, excessive filtering may lead to missed dataflow edges.

Extracting dataflow edges that refer to objects

Lienhard et al. propose an analysis that uses execution traces to extract an Object Flow Graph (OFG) in which edges refer to objects, and nodes represent classes and groups of classes [74]. The OFG analysis addresses the aliasing challenge, and extracts dataflow edges for field read, field write, and method invocation expressions in the code, the same expressions used by ScoriaX. Since nodes represent classes, an OFG is unable to show dataflow communication between different instances of the same class.

Dynamic hierarchical object graphs

A dynamic analysis does not require annotations to extract a hierarchy of objects [83]. However, dynamic analysis can infer an object hierarchy only based on strict encapsulation where a child object at the lower levels is accessible only through its parent. In Scoria, however, a parent object can have a child object that is conceptually part of it without being dominated by the parent. Although a dynamic analysis has fewer false positives than a static analysis, it finds vulnerabilities only if the architects are able to write unit tests that expose the vulnerability. The goal of security analyses is to find vulnerabilities that may not be already known. To my best knowledge, dynamic hierarchical object graphs have only been used in program comprehension and for finding performance issues [83].

7.2 Finding Security Vulnerabilities

This section describes approaches that find security vulnerabilities. ARA is one of the three pillars in finding security vulnerabilities along penetration testing and code inspection [86]. A security vulnerability can be caused by a software defect, which is local, and an approach can find such a vulnerability by analyzing one class at a time. Finding a security vulnerability that exist due to an architectural flaw requires understanding the program at a higher-level and often uses information that is not available in the code such as security properties and constraints. This chapter distinguishes between approaches that separate constraints from extraction.

7.2.1 AST-Based Static Analyses

Analyses that are fully automatic usually avoid a manual setup such as annotations [25]. They traverse the Abstract Syntax Tree (AST) of the program looking for vulnerabilities localized in a class or a method. While an AST-based analysis supports legacy code, vulnerabilities are described as patterns or AST visitors [127]. Although these pattern-based approaches provide automated detection, the output of a static analysis tool still requires human evaluation [32].

For example, consider the commercial tool SecureAssist [33] that is a security "spell checker" for Java. One SecureAssist pattern is the invocation of the method `createTempFile` of the method `File`. SecureAssist reports for example that UPMA creates a temporary file in the class `HTTPTransport`. The existence of a temporary file does not necessarily imply the existence of security vulnerability. In UPMA, the architects need to inspect how the temporary file is used in another class, `SyncDatabaseActivity`. The inspection reveals that the temporary file represents the encrypted database downloaded from a remote server. If the file is more recent than the current database file, the application overrides its content and deletes it. Since the temporary file is encrypted, no information disclosure exists.

7.2.2 Android-Specific Analyses

For Android applications, static analyses identify applications that access resources for which the user does not explicitly provide permissions, or abuse the user trust and disclose confidential information. Blue Seal [59], for example, uses an information flow analysis to generate flow permissions. The user can give an application permission to access the GPS location or the network to display advertisements, and the user trusts that the application does not disclose the current location to third party servers. If such an information flow exists, Blue Seal notifies the user before installation.

Grace et al. [52] propose a static analysis to determine resources that applications can access without having proper permissions. Rather than using an existing static analysis framework, this approach implements its own parser that keeps track of the subtyping relations and conservatively assumes that a reference passed as an argument could be of a subtype of the method formal parameter type. Since the approach is too conservative, it prunes infeasible path for scalability and models the Android permissions in method summaries. If a method checks for permission before it accesses confidential resources, the method is not analyzed. The analysis also determines public methods in applications provided by third party vendors that can bypass permissions checks and access confidential resources. For example, it finds an application on HTC phones that can send SMS messages without having the *SMS_SEND* permission enabled by the user.

7.2.3 Static Taint Analyses

Similar to Scoria, a static taint analysis [49, 126, 48] finds information disclosure and tampering by tracking how the values flow[1] from a source (a method that returns confidential information) to a sink (a method that discloses information received as arguments) without passing through a sanitizer (a method that checks if the input has a predefined format). A static taint analysis uses a SDG or an ICFG and a separate may-alias analysis provided

by a static analysis framework. Turning on sensitivity flags to increase precision may also increase the analysis time and reduce scalability [85]. To improve scalability, one solution is to use an on-demand may-alias analysis [125] that considers only the variables with assigned security properties.

FlowDroid

FlowDroid [48] is a static taint analysis that reasons about information flow at the level of variables and expressions in the code. FlowDroid considers aliasing and relies on SOOT [72] to extract an ICFG of the program. It uses an on-demand may-alias [125] such that it is object-, field-, and context-sensitive and allow the architects to increase the precision by turning on sensitivity flags. If the approach reports a false positive, the architects need to understand the inner workings of the static analysis, which is non-trivial.

Finding a vulnerability means checking the constraint that no path from sources to sinks exists in ICFG. The constraint uses transitive information flow and object reachability through points-to edges. If the architects notice a false negative, they may change the list of sources and sinks, but cannot change a constraint without changing the static analysis. The initial list of sources and sinks is automatically generated [20], and architects can alter the list as needed. Therefore, the architects have some flexibility in assigning security properties but cannot distinguish between two invocations of the same method in different contexts. FlowDroid works on Java and web-application, but it focuses on Android applications and models the state of objects of type `Activity`, `Application`, and `Service` in the Android framework by parsing configuration files.

7.2.4 Dynamic Taint Analyses

TaintDroid [42] is a dynamic taint analysis focused on Android applications that assigns security properties to values at each bytecode instruction executed. TaintDroid monitors the

[1]We use value flow instead of data flow terminology to distinguish a data flow between variables from dataflow communication between objects.

execution and stops the monitored application if a tainted value does not satisfy a given security constraint, and can find information disclosure when the device identification number or the current location are sent to a server by assigning security properties at the level of variables, methods and files. The constraints are however hard-coded such that any object that contains a tainted value is considered tainted, and are limited to simple checks to avoid the performance overhead for the approach to be practical. Another disadvantage of TaintDroid is the amount of engineering needed, which is specific not only to the version of Android framework used, but also to multiple heterogeneous devices of a variety of manufacturers on which TaintDroid is installed.

7.2.5 Commercial Security Tools

The section describes several commercial static analyses that find security vulnerabilities and coding defects. Most commercial tools include AST-based analyses that find local software defects such as buffer overflows. They provide robust static analyses that scale to large code bases as required by industry. Since the architects have a low tolerance to false positives, commercial tools often sacrifice soundness for precision, and focus on scalability.

HP Fortify SCA

Fortify Static Code Analyzer [62, 31] is a commercial tool that provides several types of analyses including AST-based analysis to find local defects and a static taint analysis to find information disclosure and tampering. The information flow analysis in Fortify is based on finding a path in the ICFG from a source to a sink based on a set of secure coding rules. Another analysis Fortify provides is a control flow analysis in which constraints restrict the invocations of methods in a certain order. The control flow analysis can find, for example, if an XML reader is properly configured before it is used.

IBM AppScan

Similarly to Fortify, AppScan Source [66] supports both an information flow analysis and a control flow analysis based on the call graph. AppScan uses a SDG and supports customizing the rules to enable the architects to extend the possible sources or sinks, or to specify methods that sanitize data or that can propagate confidential data. Although more flexible comparing to FlowDroid, the constraints have a fixed form. A constraint has a source, a sanitizer, and a sink and checks that no information flows from a source to a sink without passing through the sanitizer [126]. In addition, AppScan provides a dynamic taint analysis and allows architects to combine results of the static and dynamic taint analysis.

Coverity Security Advisor

Another commercial tool that provides an AST-based analysis and a static taint analysis is Coverity [35], which provides several rules to find security defects. The rule of Coverity that is similar to a static taint analysis is "insecure data handling" that ensures no tainted input can be used directly or indirectly to modify files or to connect to databases using SQL queries. Thus, Coverity can prevent tampering such as SQL injection.

The main advantage of Coverity is that it focuses not only on finding security vulnerabilities but also on supporting developers to fix the vulnerability once found. Coverity provides traceability to code from the identified defects to the lines of code that contain the vulnerability. The vulnerability can be scattered over several lines of code from which developers need to understand how the vulnerability may occur.

Coverity and the other commercial tools focus also on other coding defects that may indirectly lead to security vulnerabilities such as dereferencing null pointers or the use of public static non-final variables. If an attacker can trigger a null pointer exception, the exception can stop the application and lead to a denial-of-service attack.

The authors of CERT Oracle Secure Coding Standard for Java [81] have evaluated several of the commercial and open-source tools previously mentioned. Table 7.2 is compiled based

Table 7.2: Tools that provide automated detection of the CERT rules (Source: [81]).

CERT rule	Coverity	Fortify	FindBugs
EXP01-J. Never dereference null pointers	✓	✓	✓
OBJ10-J. Do not use public static nonfinal variables	✓	✓	✓
MSC03-J. Never hard code sensitive information	✓	✓	
IDS00-J. Sanitize untrusted data passed across a trust boundary	✓	✓	✓
IDS01-J. Normalize strings before validating them		✓	
IDS02-J. Canonicalize path names before validating them		✓	
IDS03-J. Do not log unsanitized user input		✓	

on CERT data and highlights those CERT rules for which the approaches provide automated detection. Most of these rules refer to coding defects as opposed to architectural flaws for which Scoria provides support (Section 6.1, page 117).

7.2.6 Type-Based Approaches

Information Flow Control

Hammer et al. [54] propose an approach, static information flow control (IFC), that is based on an object-sensitive and flow-sensitive SDG. The IFC approach finds information disclosure (assures confidentiality) and tampering (assures integrity) and provides flexibility in assigning security properties to the types of expressions, where the security properties are organized in a lattice following Bell-LaPadula model [23, 22]. The constraints are however combined in the IFC, and require non-trivial changes to support for example encryption [71].

Implementations of the approach such as JOANA [53] use the SDG generated by WALA [65] and support both explicit and implicit flows. Compared to the aforementioned taint analyses, IFC is sound and supports assigning security properties to expressions rather than method declarations. It finds for example, the vulnerability in SecretViewer which FlowDroid misses due to the unsound call graph, but reports a false positive for ACipher, which FlowDroid and Scoria avoids (Section 6.3, page 135). Since JOANA does not support Android applications, a detailed comparison with Scoria based on DroidBench remains as future work.

Security information is unavailable in the code. Some approaches [46, 37, 92] require architects to provide security types as annotations in the code. These annotations are different from the annotations Scoria uses, which focus only on describing aliasing, without providing security information.

Role-based access control

To enhance the code with an authorization mechanism such as role-based access control many approaches use Java annotations. In most cases, these annotations are only checked at runtime. In contrast, a typechecker ensures that annotations and code are consistent at compile-time. For example, Object Role Based Access Control (ORBAC) [46] uses a typechecker to enforce the security policy that a user cannot read confidential data of other users unless their roles allow him to do so. However, ORBAC does not handle object hierarchy and the role-based protection is not propagated to children objects, although the children may represent confidential data.

Type-systems for security

A type system [37, 92] allows architects to add security properties as annotations that augment the type of a variable that includes confidential value (`@Tainted`) or that can be exposed (`@Untainted`). Then, architects aided by the typechecker propagate the annotations in the code. To find information disclosure a typechecker analyzes one class at a time and ensures that no `@Tainted` is assigned to an `@Untainted` value. In Tainting Checker [37, 102], the constraint is a simple hard-coded check that the architects cannot change without redesigning the typechecker. The main advantage of a typechecker is that it only needs an AST and an intra-procedural CFG as a program representation, which makes the tainting type system modular and scalable. A tainting type system provides soundness, supports aliasing while the precision is provided by annotations.

To find the information disclosure in Fig. 7.3, `aToken` needs to be annotated `@Tainted`,

```
1  class Main<owner>{
2   domain UI,LOGIC,DATA;
3   View<UI,LOGIC,DATA> view = new View();
4  }
5  class View<owner,L,D> {
6    domain OWNED;
7    Host<L,D> h = new Host();
8    Client<L,D> c = new Client(h);
9    void onCreate(String<D> pwd){
10      c.authenticate(pwd);
11      File<D> tFile = new File("client.tmp");
12      FileOutputStream<OWNED> fos = new FileOutputStream(tfile);
13      ObjectOutputStream<OWNED> out = new ObjectOutputStream(fos);
14      out.write(c);
15   }
16 }
17 class Client<owner,D> {
18   domain OWNED;
19   String<OWNED> aToken;
20   Host<L,D> h;
21   void authenticate(String<D> pwd) {
22     aToken = h.getAccessToken(pwd);
23   }
24   void post(String<lent> msg) {
25     h.post(msg, aToken);
26   }
27 }
```

Figure 7.3: Object reachability: The client object and the access token are persistently stored. A malicious client can impersonate the original client and post messages. Security constraint: *A dataflow edge with an untrusted destination does not refer to an object from which confidential data is reachable.*

while the parameter of the method `write` is annotated `@Untainted`. To find the vulnerability, the type system still needs reachability information because only a variable of the enclosing type of `aToken` is passed as an argument to the method `write`. The type system would need to consider all types that have at least one field annotated `@Tainted` as confidential, which may be overly conservative. If the developer were to use a serialization library and marks the field `aToken` as transient then the confidential data is not saved and the type system reports a false positive. Different serialization libraries require different implementations of the type systems. In Scoria, the architects can add the *IsTransient* security property and refine the constraint to avoid the false positive.

JFlow [92] is a Java-like programming language for writing secure programs, and Jif [93] is a type system that assures confidentiality and integrity for programs written in JFlow.

However, JFlow is not backward compatible with Java, and the type annotations are more complex than the ones Scoria uses, which only declare domains or put objects in domains. JFlow supports multiple types of annotations including among others annotations for assigning security properties and for defining constraints on explicit and implicit information flow. Since Jif checks the JFlow constraint annotations one class at a time, the constraints seem to enforce information flow control only locally. It would be interesting to compare JFlow constraints with the Scoria constraints that can use global information about objects that are created in different parts of the program and have the same conceptual purpose.

7.2.7 Querying Object Graphs

Researchers proposed approaches that support querying object graphs to find software defects and architectural flaws. For example, Object Query Language (OQL) [136] queries a snapshot of a heap and allows architects to reason about reachability. Since the heap snapshot has only points-to edges, the architects cannot reason about dataflow communication. Program Query Language (PQL) [84] allows architects to reason about an object graph. Queries are written in a Java-like language and can be executed against object graphs extracted by both dynamic and static analysis. However, since the object graphs are non-hierarchical, PQL does not support queries in terms of object hierarchy.

PQL can enforce constraints such as "a password from the user should not be written in a file without encryption". Writing constraints and assigning security property is separate from extraction and the architects can write queries that are code patterns. A static checker finds all the potential matches similarly to an AST-based analysis. A dynamic analysis looks for the patterns in the execution trace as well. The static analysis part is sound and considers aliasing. It uses a flow-insensitive, but context-sensitive analysis that stores the may-alias relation in a deductive database where data is represented using binary decision diagrams (BDD). Querying may-alias relations is efficiently done using the declarative language Datalog [63].

7.2.8 Operating System-level Approaches

Although an operation system has no security vulnerabilities, developers may still introduce vulnerabilities at the application-level. For example, Mai et al. [82] propose a lightweight operating system (ExpressOS) and prove that it is free of vulnerabilities. In the evaluation, the authors consider about 300 vulnerabilities listed by the Common Weakness Enumeration Initiative (CWE) [90]. Although ExpressOS prevented all OS-level vulnerabilities, it cannot prevent some application-level architectural flaws.

7.2.9 Architectural-Level Approaches

An architectural-level approach first extracts a high-level, architectural representation from the code. For object-oriented code, the representations can be of multiple types such as a code architecture that shows how code is organized in classes and packages [91], a runtime architecture that shows objects and group of objects [5], or a deployment view that shows how components are deployed on hardware components [16].

Code architecture

To support ARA, Sohr et al. propose an architectural centric approach to find architectural flaws [115]. Scoria shares the same motivation, but the proposed solutions differ. Sohr et al. approach uses Bauhaus [108] that extracts a code architecture. In Berger et al. [24], Bauhaus approximates the runtime architecture for two web applications, which are then used to find architectural flaws. However, their approximated architecture does not distinguish between components of the same type used in different contexts.

Writing constraints In Berger et al. [24], the architects can write constraints in the Object Constraint Language (OCL) [97], which is a declarative language that is part of UML. However, the edges on the runtime architecture have no labels. This makes it difficult for the architects to write constraints about the information content of a dataflow. Indeed,

the evaluation is conducted in a style similar to Reflexion models [91], focusing only on the presence or absence of communication.

Runtime architecture

Almorsy et al. [15, 16] use various UML models such as sequence diagrams extracted with commercial reverse engineering tools [17]. One advantage of the sequence diagram over the object graph is the flow-sensitivity; i.e., the sequence diagram has the time dimension that shows the order in which methods are invoked. However, the sequence diagram is only a partial representation of the runtime architecture. Moreover, the static analysis that extracts the sequence diagram is an AST visitor that does not consider aliasing. Therefore, in the extracted sequence diagram, the object type is the same as the declared type of the reference to the object, which can be an interface.

An interface can have multiple implementation classes, and each instance of these classes can have distinct security property values. Finding precisely the type behind an interface is non-trivial. For the Listeners example (Fig. 7.1), the sequence diagram for the invocation `notifyListeners` shows one abstract object `l:Listener` for the method invocation `l.update(msg)` (line 19). At runtime, the variable `l` may alias the objects `b:BarChart` and `p:PieChart` but not `m:Model`. This information is available on the object graph, but not on the sequence diagram (Fig 7.4).

He et al. [55] formalize the software architectural model (SAM) using Petri nets. Similar to SAM, the object graph used by Scoria is hierarchical, but is less restrictive than a Petri net that requires two disjoint sets of nodes: transitions and places. Dataflow objects are similar to transitions, but with fewer restrictions, i.e., dataflow objects can also be nodes in the object graph.

Writing constraints To write constraints, architects can also use an architectural description language (ADL) such as ACME [9], which allows the architects to extend the

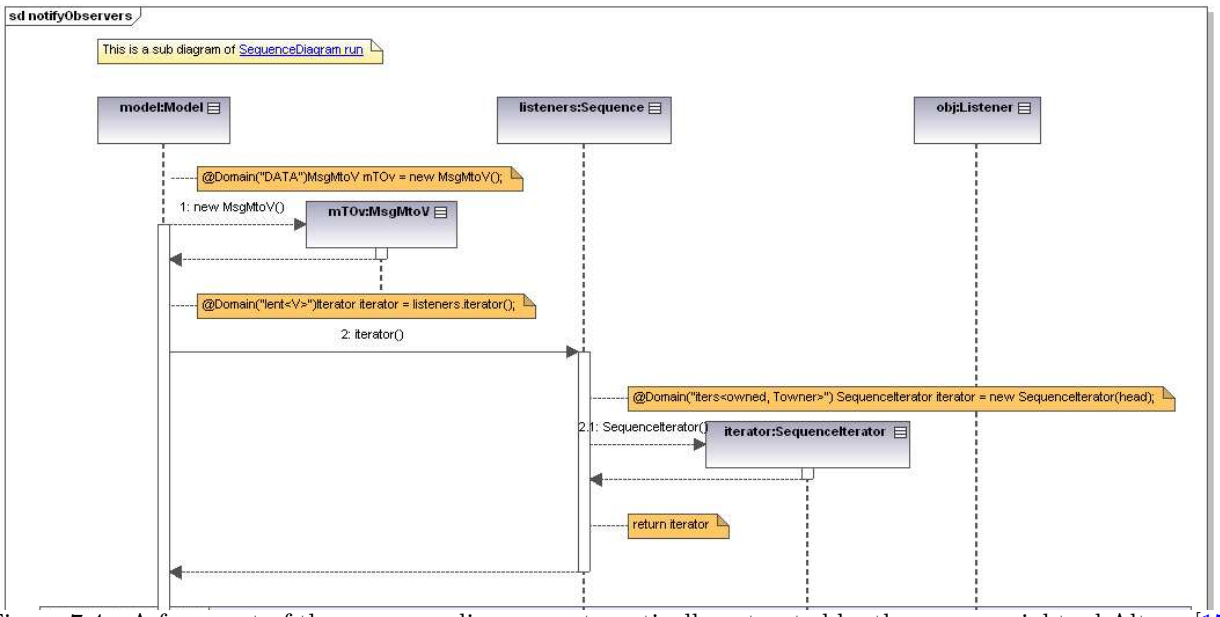


Figure 7.4: A fragment of the sequence diagram automatically extracted by the commercial tool Altova [17] for the Listeners example (Fig. 7.1).

set of properties and constraints. ACME also supports hierarchical decomposition, but an ACME connector cannot refer to a component, and hence would not allow architects to model dataflow edges that refer to objects. Rather than require architects to learn a separate language for writing constraints, a Scoria constraint is written in a Java-like language and uses assertions similar to the ones for unit testing. Although less extensible than an ADL or OCL and not declarative, Scoria still provides extensibility of properties and constraints using Java interfaces. Architects can add additional properties and constraints by implementing these interfaces.

To write OCL constraints in an IDE such as Eclipse architects and developers need to use additional plugins such as Eclipse Modeling Framework (EMF) [119]. OCL is extensible, but exporting the representation of Scoria to EMF requires a further extension because modeling a hierarchical object graphs and dataflow edges that refer to objects are not part of UML or EMF.

7.3 Evaluation of Security Approaches

This section discusses related security benchmarks and the alternatives that researchers use to evaluate approaches for finding security vulnerabilities.

7.3.1 Security Benchmarks

A test case in the benchmark has a few classes and is designed to check if an approach finds a few vulnerabilities while it avoids false positives. As an advantage over using large applications, benchmarks allow systematic evaluation and comparison of security approaches. There are other benchmarks in addition to the ones described in Chapter 6 (page 117).

Livshits et al. proposed the benchmark SecuriBench Micro [78] for evaluating approaches that find vulnerabilities in web-applications. The test cases focus on vulnerabilities such as injection attacks, and check if the static approaches handle aliasing, collections, and dataflow communication. FlowDroid performs well on SecuriBench Micro but does not find two vulnerabilities related to dataflow communication. It would be interesting to extend the evaluation of Scoria to SecuriBench Micro in addition to the evaluation on DroidBench (Section 6.3, page 135).

MalGenome [138] is a collection of 1200 malware Android applications. This thesis focuses on approaches that find vulnerabilities in legitimate well-intended applications. The approaches may not be able to detect malware, which often avoid detection by taking advantage of known limitations of static analyses.

7.3.2 Applications with Injected Vulnerabilities

Several web applications with injected vulnerabilities are available and allow evaluation of security approaches [100]. SecuriBench [79] is a collection of several web applications that span from hundreds to tens of thousands of lines of code. Each application has several injected vulnerabilities such as SQL injection that are common in web applications. The

benchmark is used by its designers to evaluate approaches such as PQL [84]. Other researchers [77] use SecuriBench to evaluate their approach. Tainting type system [37] is also evaluated using a test case from SecuriBench.

An Android application with injected vulnerabilities is InsecureBank [101]. It has several injected vulnerabilities including information disclosure to a file stored on the external memory card of the device. FlowDroid is able to find the vulnerabilities on the client side, which is an Android application, but not on the server side, which is implemented in Python.

7.3.3 Case Studies on Real-World Applications

Enck et al. [43] evaluate Fortify on popular Android applications available on the market. Fortify can find several classes of vulnerabilities including the misuse of the device identification number or the GPS location. Other approaches [42, 45] use dynamic analysis to find vulnerabilities by monitoring applications. Their goal is to identify and prevent vulnerabilities at runtime. These approaches were evaluated on hundreds of applications selected based on their popularity.

Using popular apps in the evaluation may not be enough. An approach may report the same vulnerability in applications that have a similar functionality, but miss other classes of vulnerabilities. For example, Fahl et al. [44] analyze more than 20 Android applications similar to UPMA that store encrypted password databases. These applications share the same vulnerability, where the password in clear text is disclosed to the clipboard.

Scandariato et al. [110] evaluate Fortify against a penetration testing approach on two open-source web-applications targeting the OWASP Top 10 list [99]. Although performed on a limited scale (nine participants), the controlled experiment indicates that Fortify finds more vulnerabilities than penetration testing with no significant difference in precision.

7.4 Summary

Most security approaches support legacy code and use a program representation that is a graph with various nodes and edges. The scalability is in general limited by soundness, aliasing, and precision of the extraction analysis. The research gap that Scoria fills in is sound approaches that provide extensibility in writing constraints, querying the graph to assign security properties, and fine-tuning the precision of the static analysis without sacrificing soundness. Chapter 8 discusses the limitations of Scoria and concludes.

Chapter 8 Conclusions and Future Work

Finding security vulnerabilities that an attacker can exploit to acquire confidential data or tamper with resources requires reasoning about the content of communication, not just about the presence or absence of communication. Thus, reasoning about the security of an object-oriented application requires a static analysis that is sound, and extracts dataflow edges that refer to objects. The complexity of the extracted graph can be overwhelming, and the solution this thesis uses is to organize the abstract objects hierarchically based on design intent provided by annotations. Chapter 3 proves the abstract object graph is sound, and Chapter 4 shows that architects can use a static analysis that extracts a sound, hierarchical object graph to reason about security by querying the abstract object graph and writing constraints.

One contribution of this thesis is extracting dataflow edges that refer to abstract objects using a sound analysis. Chapter 6 provides support for the hypothesis that the architects can find security vulnerabilities by executing constraints in terms of object provenance and indirect communication on an abstract object graph. These extensible constraints can find vulnerabilities that are otherwise missed by an analysis that hard-codes constraints only based on information flow that track only object transitivity and object reachability. Some of the vulnerabilities were missed due to unsoundness, while others because finding them requires reasoning about object provenance.

In the following, Section 8.1 discusses several limitations of the approach, and Section 8.2 suggests future research directions.

8.1 Limitations

The section discusses limitations of the extraction analysis, and of writing constraints.

8.1.1 Limitations of the Extraction

Potential unsoundness. Scoria does not handle reflection or dynamic code loading directly, which is a common limitation of static approaches and may lead to missing dataflow edges. In Scoria, the architects can summarize the effects of reflection using annotations by defining virtual object allocations or virtual dataflows that lead to additional objects or dataflow edges between abstract objects [47]. Another solution is to use a separate constant propagation analysis to reason about arguments of methods and classes that allow the use of Java reflection [113]. The constant propagation works well when the argument is the actual name of a class, but is more difficult when the argument is created on the fly and may be known only at runtime based on constants in configuration files or constants initialized by the users.

In this thesis, soundness is proven on a declarative specification of the analysis using inference rules, while the implementation follows closely an imperative description based on transfer functions. To avoid any discrepancies from the specification, the analysis could be implemented using declarative languages such as Datalog [63] similarly to the static analyses in DOOP [113, 114].

Precision. The extracted abstract object graph is an over-approximation of any possible execution, and, as discussed in Section 2.9.3, some of the objects or dataflow edges are false positives. The extraction analysis relies on the precision that the annotations provide. The evaluation (Section 4.4) provides examples where more precision is needed. Supported by a typechecker, the architects can refine the annotations such that the extraction analysis places objects of the same type in different groups. Therefore, the architects can improve the precision of the extraction analysis as needed in a local manner, rather than turning on sensitivity flags for the whole program.

Scalability. The thesis evaluates ScoriaX on three applications with more than 20KLOC of Java code. Therefore, Scoria scales to real-world applications of a size of an Android

application, without being Android specific. It would be interesting to evaluate Scoria on more and larger systems such as the web-applications in the benchmark SecuriBench [79]. The main bottleneck is the effort required to add annotations; a related study measured the effort of adding annotations to be 1hour/KLOC [6]. Because the annotations implement a type system, they are amenable to type inference that is a separate research problem and an active area of research [64]. Although adding annotations requires significant effort or an expensive inference analysis, this effort is required once. The constraints that find security vulnerabilities are executed separately on the extracted graph.

8.1.2 Effort of Writing Constraints

Scoria is not a push-button approach, and requires manual effort from architects that includes adding annotations, assigning security properties and writing constraints. As discussed in Section 6.3.2, separating constraints and security properties from extraction is also one of the strengths of Scoria that architects can use to increase precision and recall. One dimension that is lacking in the evaluation is measuring the effort involved in applying each of the approaches. This information is not provided in the original benchmark for FlowDroid [48]. In addition, since the Scoria evaluators on DroidBench were still learning to use the approach and the tool, we did not measure the effort. In the case of Scoria, the effort of adding annotations and writing constraints can be significant, so is worth measuring. Since Scoria outperforms other approaches in some cases only, it would be interesting to study whether the improvement in terms of fewer false negatives or higher precision is worth the extra effort.

Currently, the constraints are written in an imperative language by using unit test assertions. It would be interesting to investigate other alternatives such as writing constraints in a declarative language [63].

The thesis provides several examples of constraints that can be reused with minimal changes, mainly for assigning security properties using application-specific information.

Some of the information about security properties can be reused across systems that use the same library of framework. The constraints based on object transitivity and indirect communication are reused with minimal changes in several of the test cases in DroidBench [48]. It would be useful to allow the architects to start with a set of reusable constraints based on the automatically generated sinks and sources used by FlowDroid [20]. Then, the architects can use Scoria to focus on more expressive, application-specific constraints.

8.2 Future Work

Future research direction could improve the extraction analysis by addressing current limitations and extract additional types of edges. Another direction may involve a field study to investigate how architects use Scoria in practice. Software maintenance [30, 103] could also benefit from an abstract object graph that is sound and hierarchical.

8.2.1 Improve Extraction

The extraction uses only the information available in the source code. To support dynamic code loading, and reflection, it would be interesting to combine ScoriaX with a dynamic analysis [83]. Ideally, a hybrid approach would use both static and dynamic analysis such that it meets both the soundness and the precision requirements. Another future direction may involve incorporating a partial order in which objects and edges are created using the static execute after/before relation [68].

8.2.2 Extract Other Types of Edges

Dataflow edges reflect communication that is explicit in the code. Future work may include extraction of other information such as implicit flows due to a condition in the code that can also lead to security vulnerabilities. Implicit flows would be an addition to the existing types of edges the implementation of the analysis extracts, such that the integration

in Scoria would be straightforward.

8.2.3 Field Study on Security Architects

It would be interesting to investigate how security architects use Scoria in practice and measure their effort to extract object graphs and to write constraints. Such a study would estimate how reusable the constraints are and highlight additional types of constraints that the designers of Scoria may not have considered.

8.2.4 Abstract Object Graphs in Software Maintenance

Another research direction may investigate the effectiveness of the abstract object graphs in software maintenance where developers often use only a code structure diagram such as a class diagram. However, in a code structure diagram, some dependencies are not obvious [135] and soundness may be required for maintenance tasks such as impact analysis and change propagation during which developers investigate all possible dependencies of a changed component. This research direction was initiated by giving developers an abstract object graphs with points-to edges only and found to reduce the overall effort of software changes compared to developers who have access only to a diagram of the code structure [18].

8.3 Conclusion

Reports of security vulnerabilities in software still occur at an alarming rate [51] as attackers find new ways to circumvent the security of applications. Approaches that find security vulnerabilities need more flexibility to adapt to new types of attacks by combining the power of tool support with the insights that only architects and security experts can provide. This idea is incorporated in Architectural Risk Analysis that is adopted in industry, but the process is still mostly manual [36].

This thesis aimed to design a principled, rigorous approach to support Architectural Risk Analysis, and the project succeeded in many ways. We were able to formalize a model of dataflow communication, prove its soundness, build tools, run them on real, high-value code including server code and mobile applications, and find several serious security vulnerabilities or places where an implicit security policy is not being respected. While additional work remains, I hope to see more rigorous and better tool-supported Architectural Risk Analysis being adopted at scale to radically reduce the incidence of security vulnerabilities.

APPENDIX

Soundness Proof

We now prove Progress, Preservation and Soundness theorems.

9.1 Theorem: Dataflow Preservation (Subject Reduction)

If

$\emptyset, \Sigma, \theta \vdash e : T$

$\Sigma \vdash S$

$G = \langle DO, DD, DE \rangle$ $\qquad G \vdash_{CT,H} \Sigma$

$\emptyset, \emptyset, G \vdash_O e$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$

then

there exists $\Sigma' \supseteq \Sigma$ *and* $T' <: T$ *such that* $\emptyset, \Sigma', \theta \vdash e' : T'$ *and* $\Sigma' \vdash S'$

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

$\emptyset, \emptyset, G \vdash_O e'$

and $G \vdash_{CT,H} \Sigma'$

The Dataflow Preservation theorem extends the FDJ Type Preservation theorem (the common parts are highlighted). Those parts are proved by induction over the derivation of the FDJ evaluation relation : $e; S \rightsquigarrow e'; S'$.

Proof. We prove preservation by induction on the instrumented evaluation relation

$$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$$

The most interesting cases are Ir-New, Ir-Read (page 182), Ir-Write (page 184), and Ir-Invk (page 186).

Case Ir-New: e = `new` $C\langle\overline{\ell'.d}\rangle(\overline{v})$, and $e' = \ell$. We have:

$$\frac{\begin{array}{c} G = \langle DO, DD, DE \rangle \qquad O = C_{\texttt{this}}\langle\overline{D_O}\rangle \qquad \forall i \in 1..|\overline{\ell'.d}| \qquad G \vdash_O D_i \in findD(C_{\texttt{this}}::\ell'.d_i) \\ O_C = \langle\ C\langle\overline{D}\rangle\ \rangle \qquad \{O_C\} \subseteq DO \\ G \vdash_O dparams(C, O_C) \qquad \{(O_C, qual(\ell'.d_i)) \mapsto D_i\} \subseteq DD \\ \Upsilon, G \vdash_O ddomains(C, O_C) \\ \forall m \in \overline{md}\ mbody(m, C\langle\overline{\ell'.d}\rangle) = (\overline{x} : \overline{T},\ e_R) \\ C\langle\overline{D}\rangle \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\langle\overline{\ell'.d}\rangle\}, \Upsilon \cup \{C\langle\overline{D}\rangle\}, G \vdash_{O_C} e_R \\ \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O \texttt{new}\ C\langle\overline{\ell'.d}\rangle(\overline{v})}\ [\textsc{Df-New}]$$

$$\frac{\begin{array}{c}\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\!<\!\overline{p}\!>\!(\overline{v})] \\ G = \langle DO, DD, DE\rangle \\ \overline{p} = \overline{\ell'.d} \qquad \forall i \in 1..|\overline{\ell'.d}|\ D_i = K[\ell'_i.d_i] \\ \ell_i \in dom(H) \text{ s.t. } H[\ell_i] = O_i \qquad D_i = DD[O_i, qual(\ell'_i.d_i)] \\ O_C = \langle C\!<\!\overline{D}\!>\rangle \qquad O_C \in DO \qquad H' = H[\ell \mapsto O_C] \\ \forall(\texttt{domain}\ d_j) \in domains(C\!<\!\overline{p}\!>) \qquad D_j = DD[(O_C, C{::}d_j)] \qquad K' = K[\ell.d_j \mapsto D_j]\end{array}}{\theta \vdash \texttt{new}\ C\!<\!\overline{p}\!>\!(\overline{v}); S; H; K; L_I; L_E \rightsquigarrow_G \ell; S'; H'; K'; L_I; L_E}[\text{IR-NEW}]$$

$$\frac{\begin{array}{c}G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R\end{array}}{G \vdash_{CT,H} \Sigma}[\text{DF-SIGMA}]$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, G \vdash_O e'$

(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ — By assumption

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ — By assumption

$\forall \ell \in dom(S),\ \Sigma[\ell] = C\!<\!\overline{\ell'.d}\!>$ — Since $\Sigma \vdash S$

$H[\theta] = O_C = \langle C\!<\!\overline{D}\!>\rangle \in DO$

and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$

and $\forall d_i \in domains(C\!<\!\overline{\theta'.d}\!>)$

$K[\theta.d_i] = D_i = \langle D_{id_i}, C{::}d_i\rangle\ \{(O_C, C{::}d_i) \mapsto D_i\} \in DD$ — By DF-APPROX

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L_I[(\ell_{src}, \theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp\rangle \in DE$ — By DF-APPROX

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E_k \in L_E[(\theta, \ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp\rangle \in DE$ — By DF-APPROX

$O_C = \langle C_\ell\overline{<D>}\rangle \in DO$ — By sub-derivation of Ir-New
$S' = S[\ell \mapsto C_\ell\overline{<\ell'.d>}(\overline{v})]$ — By sub-derivation of Ir-New
$H' = H[\ell \mapsto O_C]$ — By sub-derivation of Ir-New
$\forall i \in 1..|\overline{\ell'.d}|\ D_i = K[\ell'_i.d_i]$ — By sub-derivation of Ir-New
$\forall(\texttt{domain}\ d_j) \in domains(C\overline{<\ell'.d>}) \quad D_j = DD[(O_C, C_\ell{::}d_j)]$
$K' = K[\ell.d_j \mapsto D_j]$ — By sub-derivation of Ir-New
$L'_I = L_I \quad L'_E = L_E$ — By sub-derivation of Ir-New
$\exists\Sigma' \supseteq \Sigma$ *and* $T' <: T$ *s.t.* $\emptyset, \Sigma', \theta \vdash e' : T'$ *and* $\Sigma' \vdash S'$ — By FDJ Type Preservation
$\Sigma'[\ell] = C_\ell\overline{<\ell'.d>}$ — By $\Sigma' \vdash S'$
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ — By Df-Approx
This proves (1).

$\emptyset, \emptyset, G \vdash_O e'$ — By Df-Loc, since $e' = \ell$
This proves (2).

$G \vdash_{CT,H} \Sigma$ — By assumption
$\forall \ell \in dom(S), \Sigma[\ell] = C_\ell\overline{<p>}$ — By sub-derivation of Df-Sigma
$H[\ell] = O_\ell = \langle C_\ell\overline{<D_\ell>}\rangle \in DO$ — By sub-derivation of Df-Sigma
$\forall m.\ mbody(m, C_\ell\overline{<p>}) = (\overline{x} : \overline{T},\ e_R)$ — By sub-derivation of Df-Sigma
$\{\overline{x} : \overline{T},\ \texttt{this} : C_\ell\overline{<p>}\}, \emptyset, G \vdash_{O_\ell} e_R$ — By sub-derivation of Df-Sigma
$O_C = \langle C\overline{<D>}\rangle \in DO$ — By sub-derivation of Ir-New
$S' = S[\ell \mapsto C\overline{<p>}(\overline{v})]$ — By sub-derivation of Ir-New
$H' = H[\ell \mapsto O_C]$ — By sub-derivation of Ir-New
$\emptyset, \emptyset, G \vdash_O e$ — By assumption with e, Υ below
$e = \texttt{new}\ C\overline{<\ell'.d>}(\overline{v})$, and $\Upsilon = \emptyset$

$\forall m.\ mbody(m, C\overline{<p>}) = (\overline{x} : \overline{T},\ e_R)$ — By sub-derivation of Df-New
$C\overline{<D>} \notin \Upsilon \implies$
$\{\overline{x} : \overline{T}, \texttt{this} : C\overline{<p>}\}, \Upsilon \cup \{C\overline{<D>}\}, G \vdash_{O_C} e_R$ — By sub-derivation of Df-New
$\{\overline{x} : \overline{T}, \texttt{this} : C\overline{<p>}\}, \emptyset, G \vdash_{O_C} e_R$ — By Df-Strengthening Lemma
$\forall \ell \in dom(S'), \Sigma'[\ell] = C_\ell\overline{<p>}$
  $H'[\ell] = O_\ell = \langle C_\ell\overline{<D_\ell>}\rangle \in DO$
  $\forall m.\ mbody(m, C_\ell\overline{<p>}) = (\overline{x} : \overline{T},\ e_R)$
    $\{\overline{x} : \overline{T},\ \texttt{this} : C_\ell\overline{<p>}\}, \emptyset, G \vdash_{O_\ell} e_R$ — By above
$G \vdash_{CT,H'} \Sigma'$ — By Df-Sigma with above H' and Σ'
This proves (3).

Case Ir-Read: e $= \ell.f_i$, and $e' = v_i$. We have:

$$\frac{\begin{array}{c} e_0 : C\langle\overline{p}\rangle \qquad (T_k\ f_k) \in fieldDecls(C) \\ G \vdash_O import(C\langle\overline{p}\rangle, T_k) \\ \Gamma, \Upsilon, G \vdash_O e_0 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k}[\textsc{Df-Read}]$$

$$\frac{\begin{array}{c} S[\ell] = C\langle\overline{p}\rangle(\overline{v}) \qquad fields(C\langle\overline{p}\rangle) = \overline{T}\ \overline{f} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad O_v = H[v_i] \qquad T_i \in \overline{T} \\ E = \langle O_\ell, O, O_v, Imp\rangle \in DE \qquad H; K; L_I; L_E \vdash O_v \in irLookup(T_i) \qquad L'_I = L_I[(\ell,\theta) \mapsto_\cup \{E\}] \end{array}}{\theta \vdash \ell.f_i; S; H; K; L_I; L_E \rightsquigarrow_G v_i; S; H; K; L'_I; L_E}[\text{IR-}\textsc{Read}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{p}\rangle \qquad H[\ell] = O = \langle C\langle\overline{D}\rangle\rangle \in DO \\ \forall m.\ mbody(m, C\langle\overline{p}\rangle) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\langle\overline{p}\rangle\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ — By assumption
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ — By assumption
$\forall \ell \in dom(S),\ \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$ — Since $\Sigma \vdash S$
$H[\theta] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO$
and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$
and $\forall d_i \in domains(C\langle\overline{\theta'.d}\rangle)$
$K[\theta.d_i] = D_i = \langle D_{id_i}, C::d_i\rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$ — By Df-Approx
$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$
$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$
$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$
$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$
$E'_k \in L_I[(\ell_{src}, \theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp\rangle \in DE$ — By Df-Approx
$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$
$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$
$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$
$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$
$E_k \in L_E[(\theta, \ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp\rangle \in DE$ — By Df-Approx
$S' = S, H' = H, K' = K, L'_E = L_E$ — By sub-derivation of Ir-Read
$S[\ell] = C_\ell\langle\overline{p}\rangle(\overline{v}) \quad fields(C_\ell\langle\overline{p}\rangle) = \overline{T}\ \overline{f}$ — By sub-derivation of Ir-Read
$O = H[\theta] \quad O_\ell = H[\ell] \quad O_v = H[v_i] \quad T_i \in \overline{T}$ — By sub-derivation of Ir-Read
$E' = \langle O_\ell, O, O_v, Imp\rangle \in DE \quad H; K; L_I; L_E \vdash O_v \in irLookup(T_i)$ — By sub-derivation of Ir-Read
$L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}]$ — By sub-derivation of Ir-Read

$\forall \ell_{src} \in dom(H'),\ fields(\Sigma'[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$
$\forall m.\ mtype(m, \Sigma'[\ell_{src}]) = \overline{T} \to T_R$
$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$
$\forall H'; K'; L'_I; L'_E \vdash O_k \in irLookup(T_k)$
$E'_k \in L'_I[(\ell_{src}, \theta)] = \langle H'[\ell_{src}], H'[\theta], O_k, Imp\rangle \in DE$ By above, since $\Sigma' = \Sigma$
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By Df-Approx
This proves (1).

$\emptyset, \emptyset, G \vdash_O e'$ By Df-Loc, since $e' = v_i$
This proves (2).

$G \vdash_{CT,H} \Sigma$ By assumption
$S' = S, H' = H$ By sub-derivation of Ir-Read
$G \vdash_{CT,H'} \Sigma'$ By Df-Sigma with the above H' and $\Sigma' = \Sigma$
This proves (3).

Case Ir-Write: e $= \ell.f_i = v$, and $e' = v$

We have:

$$\frac{\begin{array}{c} e_0 : C\langle\overline{p}\rangle \qquad (T_k\ f_k) \in fields(C\langle\overline{p}\rangle) \\ e_1 : C_1\langle\overline{p''}\rangle \qquad C_1\langle\overline{p''}\rangle <: T_k \\ G \vdash_O export(C\langle\overline{p}\rangle, C_1\langle\overline{p''}\rangle) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O e_1 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k = e_1}\text{[Df-Write]}$$

$$\frac{\begin{array}{c} S[\ell] = C\langle\overline{p}\rangle(\overline{v}) \qquad fields(C\langle\overline{p}\rangle) = \overline{T}\ \overline{f} \\ S' = S[\ell \mapsto C\langle\overline{p}\rangle([v/v_i]\overline{v})] \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad O_v = H[v] \qquad H; K; L_I; L_E \vdash O_v \in irLookup(T_i) \qquad T_i \in \overline{T} \\ E = \langle O, O_\ell, O_v, Exp\rangle \in DE \qquad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E\}] \end{array}}{\theta \vdash \ell.f_i = v; S; H; K; L_I; L_E \rightsquigarrow_G v; S'; H; K; L_I; L'_E}\text{[IR-Write]}$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{p}\rangle \qquad H[\ell] = O = \langle C\langle\overline{D}\rangle\rangle \in DO \\ \forall m.\ mbody(m, C\langle\overline{p}\rangle) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\langle\overline{p}\rangle\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}\text{[Df-Sigma]}$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, G \vdash_O e'$

(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ — By assumption

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ — By assumption

$\forall \ell \in dom(S),\ \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$ — Since $\Sigma \vdash S$

$H[\theta] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO$

and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$

and $\forall d_i \in domains(C\langle\overline{\theta'.d}\rangle)$

$K[\theta.d_i] = D_i = \langle D_{id_i}, C::d_i\rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$ — By Df-Approx

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L_I[(\ell_{src}, \theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp\rangle \in DE$ — By Df-Approx

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E_k \in L_E[(\theta, \ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp\rangle \in DE$ — By Df-Approx

$H' = H, K' = K, L'_I = L_I$ By sub-derivation of Ir-Write

$S[\ell] = C_\ell\overline{<p>}(\overline{v}) \quad fields(C_\ell\overline{<p>}) = \overline{T}\ \overline{f}$ By sub-derivation of Ir-Write

$S' = S[\ell \mapsto C_\ell\overline{<p>}([v/v_i]\overline{v})]$ By sub-derivation of Ir-Write

$O = H[\theta] \quad O_\ell = H[\ell] \quad O_v = H[v] \quad T_i \in \overline{T}$ By sub-derivation of Ir-Write

$E = \langle O, O_\ell, O_v, Exp\rangle \in DE \quad H; K; L_I; L_E \vdash O_v \in irLookup(T_i)$ By sub-derivation of Ir-Write

$L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E\}]$ By sub-derivation of Ir-Write

$\exists \Sigma' \supseteq \Sigma$ *and* $T' <: T$ *s.t.* $\emptyset, \Sigma', \theta \vdash e' : T'$ *and* $\Sigma' \vdash S'$ By FDJ Type Preservation

$\Sigma'[\ell] = C\overline{<\ell'.d>}$ $\Sigma' \vdash S'$

$\forall \ell_{dst} \in dom(H'),\ fields(\Sigma'[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma'[\ell_{dst}]) = \overline{T} \to T_R$

$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$\quad\quad\quad \forall O_k . H'; K'; L'_I; L'_E \vdash O_k \in irLookup(T_k)$

$\quad\quad\quad\quad E_k \in L'_E[(\theta, \ell_{dst})] = \langle H'[\theta], H'[\ell_{dst}], O_k, Exp\rangle \in DE$ By above

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By Df-Approx

This proves (1).

$\emptyset, \emptyset, G \vdash_O e'$ By Df-Loc, since $e' = v_i$

This proves (2).

$G \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell \in dom(S), \Sigma[\ell] = C_\ell\overline{<p>}$

$\quad H[\ell] = O_\ell = \langle C_\ell\overline{<D_\ell>}\rangle \in DO$

$\quad \forall m.\ mbody(m, C_\ell\overline{<p>}) = (\overline{x} : \overline{T},\ e_R)$

$\quad\quad \{\overline{x} : \overline{T},\ \texttt{this} : C_\ell\overline{<p>}\}, \emptyset, G \vdash_{O_\ell} e_R$ By sub-derivation of Df-Sigma

$H' = H$ By sub-derivation of Ir-Write

$S[\ell] = C_\ell\overline{<p>}(\overline{v}) \quad fields(C_\ell\overline{<p>}) = \overline{T}\ \overline{f}$ By sub-derivation of Ir-Write

$S' = S[\ell \mapsto C_\ell\overline{<}(>[v/v_i]\overline{v})]$ By sub-derivation of Ir-Write

$G \vdash_{CT,H'} \Sigma'$ By Df-Sigma with the above H' and $\Sigma' = \Sigma$

This proves (3).

Case Ir-Invk: e $= \ell.m(\overline{v})$, and $e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$

We have:

$$\frac{\begin{array}{c}\ell : C\langle\overline{\ell'.d}\rangle \qquad mtype(m, C\langle\overline{\ell'.d}\rangle) = \overline{T'} \to T'_R \qquad mtypeDecl(m, C) = \overline{T_f} \to T_R \\ G \vdash_O import(C\langle\overline{\ell'.d}\rangle, T_R) \\ \forall k \in 1..|\overline{v}|\ v_k : T_a \qquad T_a <: T'_k \qquad G \vdash_O export(C\langle\overline{\ell'.d}\rangle, T_a) \\ \Gamma, \Upsilon, G \vdash_O \ell \qquad \Gamma, \Upsilon, G \vdash_O \overline{v}\end{array}}{\Gamma, \Upsilon, G \vdash_O \ell.m(\overline{v})}[\textsc{Df-Invk}]$$

$$\frac{\begin{array}{c}S[\ell] = C\langle\overline{p}\rangle(\overline{v}) \qquad mbody(m, C\langle\overline{p}\rangle) = (\overline{x}, e_R) \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad mtype(m, C\langle\overline{p}\rangle) = \overline{T} \to T_R \\ H; K; L_I; L_E \vdash O_r \in irLookup(T_R) \qquad E' = \langle O_\ell, O, O_r, Imp\rangle \in DE \qquad L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}] \\ \forall k \in 1..|\overline{x}|\ O_k = H[v_k] \qquad H; K; L_I; L_E \vdash O_k \in irLookup(T_k) \qquad T_k \in \overline{T} \\ E_k = \langle O, O_\ell, O_k, Exp\rangle \in DE \qquad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E_k\}]\end{array}}{\theta \vdash \ell.m(\overline{v}); S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S; H; K; L'_I; L'_E}[\text{IR-}\textsc{Invk}]$$

$$\frac{\begin{array}{c}G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{p}\rangle \qquad H[\ell] = O = \langle C\langle\overline{D}\rangle\rangle \in DO \\ \forall m.\ mbody(m, C\langle\overline{p}\rangle) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\langle\overline{p}\rangle\}, \emptyset, G \vdash_O e_R\end{array}}{G \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, G \vdash_O e'$

(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ — By assumption

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ — By assumption

$\forall \ell \in dom(S),\ \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$ — Since $\Sigma \vdash S$

$H[\theta] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO$

and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$

and $\forall d_i \in domains(C\langle\overline{\theta'.d}\rangle)$

$K[\theta.d_i] = D_i = \langle D_{id_i}, C::d_i\rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$ — By Df-Approx

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L_I[(\ell_{src}, \theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp\rangle \in DE$ — By Df-Approx

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E_k \in L_E[(\theta, \ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp\rangle \in DE$ — By Df-Approx

$S' = S \quad H' = H \quad K' = K$ By sub-derivation of IR-INVK

$S[\ell] = C_\ell\langle\overline{p}\rangle(\overline{v}) \quad mbody(m, C_\ell\langle\overline{p}\rangle) = (\overline{x}, e_R)$ By sub-derivation of IR-INVK

$H[\theta] = O \quad H[\ell] = O_\ell$ By sub-derivation of IR-INVK

$mtype(m, C_\ell\langle\overline{p}\rangle) = \overline{T} \to T_R \quad T_R = C_R\langle\overline{p'}\rangle$ By sub-derivation of IR-INVK

$\forall O_r.H, K, L_I, L_E \vdash O_r \in irLookup(T_R)\ E' = \langle O_\ell, O, O_r, Imp\rangle \in DE$ By sub-derivation of IR-INVK

$L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}]$ By sub-derivation of IR-INVK

$\forall k \in 1..|\overline{x}|\ O_k = H[v_k] \quad H; K; L_I; L_E \vdash O_k \in irLookup(T_k) \quad T_k \in \overline{T}$ By sub-derivation of IR-INVK

$E_k = \langle O, O_\ell, O_k, Exp\rangle \in DE \quad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E_k\}]$ By sub-derivation of IR-INVK

$\exists \Sigma' \supseteq \Sigma \ \ and\ T' <: T\ s.t.\ \emptyset, \Sigma', \theta \vdash e' : T'\ and\ \Sigma' \vdash S'$ By FDJ Type Preservation

$\Sigma'[\ell] = C_\ell\langle\overline{\ell'.d}\rangle$ $\Sigma' \vdash S'$

$\forall \ell_{src} \in dom(H'),\ fields(\Sigma'[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

$\forall m.\ mtype(m, \Sigma'[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$\forall O_k.H'; K'; L'_I; L'_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L'_I[(\ell_{src}, \theta)] = \langle H'[\ell_{src}], H'[\theta], O_k, Imp\rangle \in DE$ By above

$\forall \ell_{dst} \in dom(H'),\ fields(\Sigma'[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\forall m.\ mtype(m, \Sigma'[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$\forall O_k.H'; K'; L'_I; L'_E \vdash O_k \in irLookup(T_k)$

$E_k \in L'_E[(\theta, \ell_{dst})] = \langle H'[\theta], H'[\ell_{dst}], O_k, Exp\rangle \in DE$ By above

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By DF-APPROX

This proves (1).

$\emptyset, \emptyset, G \vdash_O e$	By assumption
$e = \ell.m(\overline{v}) \quad e_0 = \ell \quad \overline{e} = \overline{v}$	By assumption
$e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$	By assumption
$\emptyset, \Sigma, \theta \vdash e : T$	By assumption
$\exists \Sigma' \supseteq \Sigma$ *and* $T' <: T$ *s.t.* $\emptyset, \Sigma', \theta \vdash e' : T'$ *and* $\Sigma' \vdash S'$	By FDJ Type Preservation
$e_0 : T_0 \quad T_0 = C_\ell\!<\overline{p}>$	By sub-derivation of Df-Invk
$mtype(m, C_\ell\!<\overline{p}>) = \overline{T} \to T_R$	By sub-derivation of Df-Invk
$\emptyset, \emptyset, G \vdash_O e_0$	By sub-derivation of Df-Invk
$\emptyset, \emptyset, G \vdash_O \overline{e}$	By sub-derivation of Df-Invk
$\{\overline{x} : \overline{T}, \texttt{this} : C_\ell\!<\alpha, \overline{\beta}>\}, \Sigma, \theta \vdash e_R : T_R \quad T_R <: T$	By FDJ *Meth*OK:
$S[\ell] = C_\ell\!<d, \overline{d'}>(\overline{v})$	By sub-derivation of Ir-Invk
$mbody(m, C_\ell\!<d, \overline{d'}>) = (\overline{x}, e_R)$	By sub-derivation of Ir-Invk
$\Sigma[\ell] = C_\ell\!<d, \overline{d'}> = T_0$	Since $e_0 = \ell$, by T-Store
$e_0 : C_\ell\!<d, \overline{d'}>$	Since $e_0 = \ell$, by T-Store
$mtype(m, C_\ell\!<d, \overline{d'}>) = \overline{T} \to T_R$	Since $e_0 = \ell$, by T-Store
$\overline{v} : \overline{T_a}$	By inversion
$\overline{T_a} <: [\overline{v}/\overline{x}, \ell/\texttt{this}]\overline{T}$	For some $\overline{T_a}$ and $\overline{T}$
there are some $D\!<\overline{d}>$ and T'_R so that:	By Method Lemma
$T'_R <: T_R$ and $C_\ell\!<d, \overline{d'}> <: D\!<\overline{d}>$	By Method Lemma
so that $\{\overline{x} : \overline{T}, \texttt{this} : D\!<\overline{d}>\}, \Sigma, \theta \vdash e_R : T'_R$	By Method Lemma
there exists $T_S, T_S <: T'_R$ s. t. $[\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T_S$	Since term substitution preserves typing
$T_S <: T'_R$ and $T'_R <: T_R$	By above
$T_S <: T_R$	By transitivity of $<:$
Take $T = T' = T_R$ in FDJ Preservation	

$\{\overline{x} : \overline{T}, \texttt{this} : C_\ell\!<d, \overline{d'}>\}, \emptyset, G \vdash_{O_C} e_R$	By Df-Sigma
$O_C = H[\ell]$	By Df-Sigma
$\emptyset, \emptyset, G \vdash_O \ell$	By Df-Loc
$\emptyset, \emptyset, G \vdash_{O_C} [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$	By Df-Substitution Lemma
$\emptyset, \emptyset, G \vdash_O \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$	By Df-Context
This proves (2).	

$G \vdash_{CT,H} \Sigma$	By assumption
$S' = S, H' = H$	By sub-derivation of Ir-Invk
$G \vdash_{CT,H'} \Sigma'$	By Df-Sigma with the above $H' = H$ and $\Sigma' = \Sigma$
This proves (3).	

Case Ir-Context: e $= \ell \triangleright v$, and $e' = v$

We have:

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, G \vdash_{O_C} e}{\Gamma, \Upsilon, G \vdash_{O,H} \ell \triangleright e}[\text{Df-Context}]$$

$$\frac{}{\theta \vdash \ell \triangleright v; S; H; K; L_I; L_E \rightsquigarrow_G v;\ S; H; K; L_I; L_E}[\text{IR-Context}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\text{<}\overline{p}\text{>} \qquad H[\ell] = O = \langle C\text{<}\overline{D}\text{>}\rangle \in DO \\ \forall m.\ mbody(m, C\text{<}\overline{p}\text{>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\text{<}\overline{p}\text{>}\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\text{Df-Sigma}]$$

$$\frac{}{\Gamma, \Upsilon, G \vdash_O \ell}[\text{Df-Loc}]$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$	By assumption
$S' = S, H' = H, K' = K, L'_I = L_I, L'_E = L_E$	By sub-derivation of Ir-Context
This proves (1).	
$\emptyset, \emptyset, G \vdash_O e'$	By Df-Loc, since $e' = v$
This proves (2).	
$G \vdash_{CT,H} \Sigma$	By assumption
$S' = S, H' = H$	By sub-derivation of Ir-Context
$G \vdash_{CT,H'} \Sigma'$	Take $\Sigma' = \Sigma$
This proves (3).	

Case Irc-New: e $= \texttt{new}\ C\text{<}\overline{p}\text{>}(v_{1..i-1}, e_i, e_{i+1..n})$, and $e' = \texttt{new}\ C\text{<}\overline{p}\text{>}(v_{1..i-1}, e'_i, e_{i+1..n})$.

We have:

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad O = C_{\texttt{this}}\text{<}\overline{D_O}\text{>} \qquad \forall i \in 1..|\overline{p}| \qquad G \vdash_O D_i \in findD(C_{\texttt{this}}::p_i) \\ O_C = \langle\ C\text{<}\overline{D}\text{>}\ \rangle \qquad \{O_C\} \subseteq DO \\ G \vdash_O dparams(C, O_C) \qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD \\ \Upsilon, G \vdash_O ddomains(C, O_C) \\ \forall m \in \overline{md}\ mbody(m, C\text{<}\overline{p}\text{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\text{<}\overline{D}\text{>} \notin \Upsilon \implies \{\overline{x} : \overline{T},\ \texttt{this} : C\text{<}\overline{p}\text{>}\}, \Upsilon \cup \{C\text{<}\overline{D}\text{>}\}, G \vdash_{O_C} e_R \\ \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O \texttt{new}\ C\text{<}\overline{p}\text{>}(\overline{e})}[\text{Df-New}]$$

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E}{\begin{array}{l}\theta \vdash \texttt{new}\ C\text{<}\overline{p}\text{>}(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ \quad \texttt{new}\ C\text{<}\overline{p}\text{>}(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E\end{array}}[\text{Irc-New}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\text{<}\overline{p}\text{>} \qquad H[\ell] = O = \langle C\text{<}\overline{D}\text{>}\rangle \in DO \\ \forall m.\ mbody(m, C\text{<}\overline{p}\text{>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\text{<}\overline{p}\text{>}\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\text{Df-Sigma}]$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$

(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-NEW
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-NEW
$\emptyset, \emptyset, G \vdash_O e_i'$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O \texttt{new}\ C\!<\!\overline{p}\!>(v_{1..i-1}, e_i', e_{i+1..n})$ By DF-NEW
This proves (2).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-NEW
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Read: e = $e_0.f_k$, and $e' = e_0'.f_k$.
We have:

$$\frac{e_0 : C\!<\!\overline{p}\!> \qquad (T_k\ f_k) \in fieldDecls(C) \qquad G \vdash_O import(C\!<\!\overline{p}\!>, T_k) \qquad \Gamma, \Upsilon, G \vdash_O e_0}{\Gamma, \Upsilon, G \vdash_O e_0.f_k}\text{[DF-READ]}$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\theta \vdash e_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G e_0'.f_i; S'; H'; K'; L_I'; L_E'}\text{[IRC-READ]}$$

$$\frac{G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \qquad \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R}{G \vdash_{CT,H} \Sigma}\text{[DF-SIGMA]}$$

To Show:
(1) $(S', H', K', L_I', L_E') \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-READ
$(S', H', K', L_I', L_E') \sim (G)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-READ
$\emptyset, \emptyset, G \vdash_O e_0'$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O e_0'.f_k$ By DF-READ
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-READ
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Write-Rcv: e = $(e_0.f_k = e_1)$, and $e' = (e'_0.f_k = e_1)$.

We have:

$$\frac{\begin{array}{c} e_0 : C\overline{<p>} \qquad (T_k\ f_k) \in fields(C\overline{<p>}) \\ e_1 : C_1\overline{<p''>} \qquad C_1\overline{<p''>} <: T_k \\ G \vdash_O export(C\overline{<p>}, C_1\overline{<p''>}) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O e_1 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k = e_1}[\textsc{Df-Write}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E}{\begin{array}{c} \theta \vdash e_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ e'_0.f_i = e_1; S'; H'; K'; L'_I; L'_E \end{array}}[\textsc{Irc-WriteRcv}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\overline{<p>} \qquad H[\ell] = O = \langle C\overline{<D>}\rangle \in DO \\ \forall m.\ mbody(m, C\overline{<p>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\overline{<p>}\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, G \vdash_O e'$

(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$	By sub-derivation of Irc-Write-Rcv
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$	By induction hypothesis
This proves (1).	

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$	By sub-derivation of Irc-Write-Rcv
$\emptyset, \emptyset, G \vdash_O e'_0$	By induction hypothesis
$\emptyset, \emptyset, G \vdash_O e_1$	By Df-Write
$\emptyset, \emptyset, G \vdash_O e'_0.f_k = e_1$	By Df-Write
This proves (2).	

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$	By sub-derivation of Irc-Write-Rcv
$G \vdash_{CT,H'} \Sigma'$	By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).	

Case Irc-Write-Arg: e = $(v.f_k = e_1)$, and $e' = (v.f_k = e'_1)$.

We have:

$$\frac{\begin{array}{c} e_0 : C\overline{<p>} \qquad (T_k\ f_k) \in fields(C\overline{<p>}) \\ e_1 : C_1\overline{<p''>} \qquad C_1\overline{<p''>} <: T_k \\ G \vdash_O export(C\overline{<p>}, C_1\overline{<p''>}) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O e_1 \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.f_k = e_1}[\textsc{Df-Write}]$$

$$\frac{\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E}{\begin{array}{c} \theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ v.f_i = e'_1; S'; H'; K'; L'_I; L'_E \end{array}}[\textsc{Irc-Write-Arg}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\overline{<p>} \qquad H[\ell] = O = \langle C\overline{<D>}\rangle \in DO \\ \forall m.\ mbody(m, C\overline{<p>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\overline{<p>}\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E$ By sub-derivation of IRC-WRITE-ARG
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E$ By sub-derivation of IRC-WRITE-ARG
$\emptyset, \emptyset, G \vdash_O e'_1$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O e'_0.f_k = e'_1$ By DF-WRITE
This proves (2).

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E$ By sub-derivation of IRC-WRITE-ARG
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-RecvInvk: e = $e_0.m(\overline{e})$, and $e' = e'_0.m(\overline{e})$.
We have:

$$\frac{\begin{array}{c} e_0 : C\!<\!\overline{p}\!> \qquad mtype(m, C\!<\!\overline{p}\!>) = \overline{T'} \to T'_R \qquad mtypeDecl(m, C) = \overline{T_f} \to T_R \\ G \vdash_O import(C\!<\!\overline{p}\!>, T_R) \\ \forall k \in 1..|\overline{e}|\ e_k : T_a \qquad T_a <: T'_k \qquad G \vdash_O export(C\!<\!\overline{p}\!>, T_a) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.m(\overline{e})}\text{[DF-INVK]}$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E}{\begin{array}{c} \theta \vdash e_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G \\ e'_0.m(\overline{e}); S'; H'; K'; L'_I; L'_E \end{array}}\text{[IRC-RECVINVK]}$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}\text{[DF-SIGMA]}$$

To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-RECVINVK
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-RECVINVK
$\emptyset, \emptyset, G \vdash_O e_0'$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O \overline{e}$ By DF-INVK
$\emptyset, \emptyset, G \vdash_O e_0'.m(\overline{e})$ By DF-INVK
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of IRC-RECVINVK
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Case Irc-ArgInvk: e $= v.m(v_{1..i-1}, e_i, e_{i+1..n})$,and $e' = v.m(v_{1..i-1}, e_i', e_{i+1..n})$.
We have:

$$\frac{\begin{array}{c} e_0 : C\!<\!\overline{p}\!> \qquad mtype(m, C\!<\!\overline{p}\!>) = \overline{T'} \to T_R' \qquad mtypeDecl(m, C) = \overline{T_f} \to T_R \\ G \vdash_O import(C\!<\!\overline{p}\!>, T_R) \\ \forall k \in 1..|\overline{e}|\ e_k : T_a \qquad T_a <: T_k' \qquad G \vdash_O export(C\!<\!\overline{p}\!>, T_a) \\ \Gamma, \Upsilon, G \vdash_O e_0 \qquad \Gamma, \Upsilon, G \vdash_O \overline{e} \end{array}}{\Gamma, \Upsilon, G \vdash_O e_0.m(\overline{e})}\text{[DF-INVK]}$$

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ v.m(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'; L_I'; L_E'\end{array}}\text{[IRC-ARGINVK]}$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}\text{[DF-SIGMA]}$$

To Show:
(1) $(S', H', K', L_I', L_E') \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-ArgInvk
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-ArgInvk
$\emptyset, \emptyset, G \vdash_O e_i'$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O v.m(v_{1..i-1}, e_i', e_{i+1..n})$ By Df-Invk
This proves (2).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-ArgInvk
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Context: e $= \ell \triangleright e_0$, and $e' = \ell \triangleright e_0'$.
We have:

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, G \vdash_{O_C} e}{\Gamma, \Upsilon, G \vdash_{O,H} \ell \triangleright e}[\text{Df-Context}]$$

$$\frac{\ell \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L_I'; L_E'}{\theta \vdash \ell \triangleright e; S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright e'; S'; H'; K'; L_I'; L_E'}[\text{Irc-Context}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \qquad \forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \qquad H[\ell] = O = \langle C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, G \vdash_O e_R \end{array}}{G \vdash_{CT,H} \Sigma}[\text{Df-Sigma}]$$

To Show:
(1) $(S', H', K', L_I', L_E') \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, G \vdash_O e'$
(3) $G \vdash_{CT,H'} \Sigma'$

$\ell \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\ell \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$O_\ell = H[\ell]$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_{O_\ell} e_0'$ By induction hypothesis
$\emptyset, \emptyset, G \vdash_O \ell \triangleright e_0'$ By Df-Context
This proves (2).

$\ell \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$G \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

□

9.2 Theorem: Dataflow Progress

$$
\begin{array}{l}
\mathit{If} \\
\quad \emptyset, \Sigma, \theta \vdash e : T \\
\quad \Sigma \vdash S \\
\quad G \vdash_{CT,H} \Sigma \\
\quad \emptyset, \emptyset, G \vdash_O e \\
\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
\mathit{then} \\
\quad \mathit{either}\ e\ \mathit{is\ a\ value} \\
\quad \mathit{or\ else}\ \theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E
\end{array}
$$

Proof. We prove progress by derivation of $\emptyset, \emptyset, G \vdash_O e$, with a case analysis on the last typing rule used. The most interesting cases are Df-New, Df-Read (page 198), Df-Write (page 201), and Df-Invk (page 203).

Case DF-NEW : e = *new* $C\overline{<p>}(\overline{e})$.

Subcase $\overline{e} = \overline{v}$ that is $e = new\ C\overline{<p>}(\overline{v})$. Take $e' = \ell$, then Ir-New can apply.

$$
\frac{\begin{array}{c}
\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\overline{<p>}(\overline{v})] \\
G = \langle DO, DD, DE \rangle \\
\overline{p} = \overline{\ell'.d} \qquad \forall i \in 1..|\overline{\ell'.d}|\ D_i = K[\ell'_i.d_i] \\
\ell_i \in dom(H) \text{ s.t. } H[\ell_i] = O_i \qquad D_i = DD[O_i, qual(\ell'_i.d_i)] \\
O_C = \langle C\overline{<D>} \rangle \qquad O_C \in DO \qquad H' = H[\ell \mapsto O_C] \\
\forall(\texttt{domain}\ d_j) \in domains(C\overline{<p>}) \qquad D_j = DD[(O_C, C{::}d_j)] \qquad K' = K[\ell.d_j \mapsto D_j]
\end{array}}{\theta \vdash \texttt{new}\ C\overline{<p>}(\overline{v}); S; H; K; L_I; L_E \rightsquigarrow_G \ell; S'; H'; K'; L_I; L_E}[\text{Ir-New}]
$$

$$
\frac{\Sigma[\ell] = C\overline{<\ell''.d>} \qquad d \in domains(C\overline{<\ell''.d>})}{\Gamma; \Sigma; \theta \vdash qual(\ell.d) = C{::}d}
$$

To show:
(1) $\forall i \in 1..|\overline{\ell'.d}| \quad D_i = K[\ell'_i.d_i]$
(2) $O_C = \langle C\overline{<D>} \rangle \quad O_C \in DO$
(3) $\forall i \in 1..|\overline{\ell'.d}| \quad H[\ell'_i] = O_i \quad D_i = DD[(O_i, qual(\ell'.d_i))]$
(4) $\forall d_j \in domains(C\overline{<\ell'.d>}) \quad D_j = DD[(O_C, C{::}d_j)]$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$\forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$ $\Sigma \vdash S$

$\forall \ell \in dom(S), \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$

$\Longrightarrow$

$H[\ell] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO$

and $\forall \ell'_j.d_j \in \overline{\ell'.d}\ K[\ell'_j.d_j] = D_j = \langle D_{id_j}, qual(\ell'_j.d_j)\rangle \in rng(DD)$

and $\forall d_i \in domains(C\langle\overline{\ell'.d}\rangle)$

$K[\ell.d_i] = D_i = \langle D_{id_i}, C{::}d_i\rangle\ \{(O_C, C{::}d_i) \mapsto D_i\} \in DD$

and $\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L_I[(\ell_{src}, \ell)]\ E'_k = \langle H[\ell_{src}], H[\ell], O_k, Imp\rangle \in DE$

and $\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E_k \in L_E[(\ell, \ell_{dst})]\ E_k = \langle H[\ell], H[\ell_{dst}], O_k, Exp\rangle \in DE$

This proves (1,2).

$\emptyset, \emptyset, G \vdash_O \texttt{new}\ C\langle\overline{\ell'.d}\rangle(\overline{v})$ By assumption

$\forall i \in 1..|\overline{\ell'.d}|\quad G \vdash_O D_i \in findD(C_{this}{::}\ell'_i.d_i)$ By sub-derivation of Df-New

$O_C = \langle\ C\langle\overline{D}\rangle\ \rangle\quad \{O_C\} \subseteq DO$ By sub-derivation of Df-New

$G \vdash_O dparams(C, O_C)$ By sub-derivation of Df-New

$\{(O_C, C{::}\alpha_i) \mapsto D_i\} \subseteq DD\quad \alpha_i \in params(C)$ By sub-derivation of AuxAlpha

$\{(O_C, qual(\ell'_i.d_i)) \mapsto D_i\} \subseteq DD$ By sub-derivation of Df-New

$\emptyset, G \vdash_O ddomains(C, O_C)$ By sub-derivation of Df-New

This proves (4). By Df-Domains Lemma

Sub-subcase $\ell'_i \neq \theta$, $\ell'_i.d_i$ is a domain of some object in the context of θ That is, $\exists\ n \in dom(H)$ such that :

$DD[(H[n], C_n{::}d_i)] = D_i$

$G \vdash_O H[n] \in lookup(\Sigma[n])$ by subderivation of Aux-Find-Public

This proves (3). Take $\ell_i = n$

Sub-subcase $\ell'_i \neq \theta$, $\ell'_i.d_i$ substitutes the j^{th} formal domain parameters of C_{this}, where $\Sigma[\theta] = C_{this}<\overline{\ell_{this}.d}>$ That is:

$O = \langle C_{this}<\overline{D'}>\rangle$
$DD[(O, C_{this}::\alpha_i)] = D'_j$
$DD[(O, qual(\ell'_i.d_i))] = D'_j$ by subderivation of Df-New

$\forall i \in 1..|\overline{\ell'.d}| \quad G \vdash_O D_i \in findD(C_{this}::\ell'_i.d_i)$ By sub-derivation of Df-New
$D_i = DD[(O, qual(\ell'_i.d_i)] = DD[(O, C_{this}::\alpha_j)] = D'_j$ By sub-derivation of Aux-FindD
This proves (3). Take $\ell_i = \theta$

Sub-subcase $\ell'_i = \theta$, $\ell'_i.d_i$ is a domain of θ

$G \vdash_O D_i \in findD(C_{this}::\ell'_i.d_i)$ By above
$D_i = DD[(O, C::d_i))]$ By sub-derivation of Aux-FindThis
$d_i \in domains(\Sigma[\ell_i])$
$qual(\ell'_i.d_i) = C::d_i$ By inversion of Qual-Var
$D_i = DD[(O, qual(\ell'_i.d_i))]$ By above
This proves (3). Take $\ell_i = \theta$

Sub-subcase $d_i =$ ::`shared`

$G \vdash_O D_i \in findD(C_{this}::\ell'_i.d_i)$ By above
$D_i = D_{\texttt{shared}} = DD[(O_{world}, ::\texttt{shared})]$ By sub-derivation of Aux-FindShared
This proves (3).

Subcase $e = new\ C<\overline{p}>(v_{1..i-1}, e_i, e_{i+1..n})$. Then Irc-New can apply.

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E}{\begin{array}{l}\theta \vdash \texttt{new}\ C<\overline{p}>(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ \quad \texttt{new}\ C<\overline{p}>(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E\end{array}}[\text{IRC-New}]$$

$\Gamma, \Upsilon, G \vdash_O e_i$ By sub-derivation of Df-New
$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By induction hypothesis
$\theta \vdash \texttt{new}\ C<\overline{p}>(v_{1..i-1}, e_i, e_{i+1..n}) S; H; K; L_I; L_E \rightsquigarrow_G$
$\quad \texttt{new}\ C<\overline{p}>(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E$ By inversion of Irc-New
This proves the case.

Case DF-VAR : $e = x$.
Not applicable since variable is not a closed term.

Case DF-LOC : $e = \ell$.
e is a value.

Case DF-READ : e = $e_0.f_i$. There are two subcases to consider depending on whether the receiver e_0 is a value.

Subcase $e_0 = \ell$. Then $e = \ell.f_i$

From IR-READ:

$$\frac{\begin{array}{c} S[\ell] = C\langle\overline{p}\rangle(\overline{v}) \qquad fields(C\langle\overline{p}\rangle) = \overline{T}\ \overline{f} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad O_v = H[v_i] \qquad T_i \in \overline{T} \\ E = \langle O_\ell, O, O_v, Imp\rangle \in DE \qquad H;K;L_I;L_E \vdash O_v \in irLookup(T_i) \qquad L'_I = L_I[(\ell,\theta) \mapsto_\cup \{E\}] \end{array}}{\theta \vdash \ell.f_i; S; H; K; L_I; L_E \rightsquigarrow_G v_i; S; H; K; L'_I; L_E}$$

To show:
(1) $O = H[\theta]$
(2) $O_\ell = H[\ell]$
(3) $O_v = H[v_i] \quad T_i \in \overline{T} \quad H,K,L_I,L_E \vdash O_v \in irLookup(T_i) \quad E = \langle O_\ell, O, C_v, Imp\rangle \in DE$

$G \vdash_{CT,H} \Sigma$	By assumption
$\forall \ell' \in dom(S), \Sigma[\ell'] = C'\langle\overline{p}\rangle$	By sub-derivation of DF-SIGMA
$H[\ell'] = O' = \langle C'\langle\overline{D'}\rangle\rangle \in DO$	By sub-derivation of DF-SIGMA
$H[\theta] = O = \langle C\langle\overline{D}\rangle\rangle \in DO$	Since $\theta \in dom(S)$
$H[\ell] = O_\ell = \langle C_\ell\langle\overline{D_\ell}\rangle\rangle \in DO$	Since $\ell \in dom(S)$
$H[v_i] = O_v = \langle C_v\langle\overline{D_v}\rangle\rangle \in DO$	Since $v \in dom(S)$
this proves (1), and (2).	

$(S,H,K,L_I,L_E) \sim (DO,DD,DE)$	By assumption
$\forall \ell \in dom(S),\ \Sigma[\ell] = C\langle\overline{\ell'.d}\rangle$	Since $\Sigma \vdash S$
$H[\theta] = O_C = \langle C\langle\overline{D}\rangle\rangle \in DO$	
and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$	
and $\forall d_i \in domains(C\langle\overline{\theta'.d}\rangle)$	
$K[\theta.d_i] = D_i = \langle D_{id_i}, C::d_i\rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$	By DF-APPROX
$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$	
$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$	
$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$	
$H;K;L_I;L_E \vdash O_k \in irLookup(T_k)$	
$E'_k \in L_I[(\ell_{src},\theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp\rangle \in DE$	By DF-APPROX
$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$	
$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$	
$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$	
$H;K;L_I;L_E \vdash O_k \in irLookup(T_k)$	
$E_k \in L_E[(\theta,\ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp\rangle \in DE$	By DF-APPROX

$\emptyset, \emptyset, G \vdash_O \ell.f_i$ By assumption

$fields(\Sigma[\ell]) = \overline{T'}\ \overline{f}$ By FDJ T-Store

Since $e_0 = \ell \in dom(H)$

$\ell : \Sigma[\ell] = C_\ell\text{<}\overline{p}\text{>}\ (T'_i\ f_i) \in fieldDecls(C_\ell)$ By sub-derivation of Df-Read

$G \vdash_O import(\Sigma[\ell], T'_i)$ By sub-derivation of Df-Read

$\emptyset, \emptyset, G \vdash_O \ell$ By sub-derivation of Df-Read

$G \vdash_O O_i \in lookup(\Sigma[\ell])$ By sub-derivation of Aux-Import

$G \vdash_{O_i} O_j \in lookup(T'_i)$ By sub-derivation of Aux-Import

$\langle O_i, O, O_j, Imp\rangle \subseteq DE$ By sub-derivation of Aux-Import

$\emptyset, \Sigma, \theta \vdash \ell : \Sigma[\ell]$ By hypothesis

$\Sigma[\ell] = C_\ell\text{<}\overline{\ell'.d}\text{>} \quad H[\ell] = O_\ell = \langle C_\ell\text{<}\overline{D_\ell}\text{>}\rangle \in DO$

$G \vdash_{H[\theta]} H[\ell] \in lookup(\Sigma[\ell])$ By Lookup Lemma

$S[\ell] = C_\ell\text{<}\overline{\ell'.d}\text{>}(\overline{v}), \Sigma[v] = C_v\text{<}\overline{v'.d}\text{>} \quad H[v_i] = O_v = \langle C_v\text{<}\overline{D_v}\text{>}\rangle \in DO$

$T'_i = C'_v\text{<}\overline{p}\text{>} \quad C_v <: C'_v$ By sub-derivation of Df-Read

To show

$\forall p_k \in \overline{p} \quad G \vdash_{H[\ell]} D'_{vk} \in findD(C_\ell{::}p_k) \quad D'_{vk} = D_{vk} = K[v'_k.d_k]$

p_k is a domain parameter, local domain of ℓ or `shared`. Therefore we split the proof in cases

Case domain parameter: $p_k = \alpha_j$

$$\exists \ell'_j.d_j \in \overline{\ell'.d} \text{ s.t } K[v'_k.d_k] = K[\ell'_j.d_j] = D_j$$

$$D_{vk} = D_j = DD[(H[\ell], C_\ell{::}\alpha_j)] = DD[(H[\ell], qual(\ell'_j.d_j))] = K[\ell'_j.d_j] \qquad \text{By Params Lemma}$$

Case local domain: $p_k = \ell.d$

$$\exists \ell'_j.d_j \in \overline{\ell'.d} \text{ s.t } K[v'_k.d_k] = K[\ell'_j.d_j] = D_j$$

$$D_{vk} = D_j = DD[(H[\ell], C_\ell{::}d_j)] = K[\ell.d_j] \qquad \text{By sub-derivation of Df-New}$$

Case local domain: $p_k = \texttt{shared}$

$$D_{vk} = D_{\texttt{shared}} = DD[(H[\ell], {::}\texttt{shared})]$$

$G \vdash_{H[\ell]} D_{vk} \in findD(C_\ell{::}qual(v'_k.d_k))$ By inversion of AuxFindD

$G \vdash_{H[\ell]} H[v_i] \in lookup(T'_i)$ By inversion of Aux-Lookup.Take $O_i = H[\ell]$

$\langle H[\ell], H[\theta], H[v], Imp\rangle \subseteq DE$ By above. Take $O = H[\theta]\ O_i = H[\ell]\ O_j = H[v_i]$

$H[v] = O_{C_v} = \langle C_v{<}\overline{D}{>}\rangle \in DO$
and $\forall v'_j.d_j \in \overline{v'.d}\ K[v'_j.d_j] = D_j = \langle D_{id_j}, qual(v'_j.d_j)\rangle \in rng(DD)$
and $\forall d_i \in domains(C_v{<}\overline{v'.d}{>})$
$K[v.d_i] = D_i = \langle D_{id_i}, C_v{::}d_i\rangle\ \{(O_{C_v}, C_v{::}d_i) \mapsto D_i\} \in DD$ By Df-Approx
$H; K; L_I; L_E \vdash H[v] \in irLookup(\Sigma[v])$ By inversion of IR-Lookup
Take $T_k = T'_i \in \overline{T'}$, this proves (3).

Subcase $e_0 = e'_0.f_i$. That is, e_0 is not a value
From Irc-Read:

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E}{\theta \vdash e_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_0.f_i; S'; H'; K'; L'_I; L'_E}$$

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis
$\theta \vdash e'_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G e''_0.f_i; S'; H'; K'; L'_I; L'_E$ By Irc-Read
Take $e' = e''_0.f_i$.

Case DF-WRITE : e = $(e_0.f_i = e_1)$. There are three subcases to consider depending on whether the receiver e_0, and e_1 are values.

Subcase $e_0 = \ell$, and $e_1 = v$. Then $e = (\ell.f_i = v)$

From Ir-Write

$$\frac{\begin{array}{c} S[\ell] = C{<}\overline{p}{>}(\overline{v}) \qquad fields(C{<}\overline{p}{>}) = \overline{T}\ \overline{f} \\ S' = S[\ell \mapsto C{<}\overline{p}{>}([v/v_i]\overline{v})] \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad O_v = H[v] \qquad H;K;L_I;L_E \vdash O_v \in irLookup(T_i) \qquad T_i \in \overline{T} \\ E = \langle O, O_\ell, O_v, Exp\rangle \in DE \qquad L'_E = L_E[(\theta,\ell) \mapsto_\cup \{E\}] \end{array}}{\theta \vdash \ell.f_i = v; S; H; K; L_I; L_E \rightsquigarrow_G v; S'; H; K; L_I; L'_E}$$

To show:
(1) $O = H[\theta]$
(2) $O_\ell = H[\ell]$
(3) $O_v = H[v_i]\ H;K;L_I;L_E \vdash O_v \in irLookup(T_i)\ E = \langle O, O_\ell, O_v, Exp\rangle \in DE$

$G \vdash_{CT,H} \Sigma$	By assumption
$\forall \ell' \in dom(S), \Sigma[\ell'] = C'{<}\overline{p}{>}$	By sub-derivation of Df-Sigma
$H[\ell'] = O' = \langle C'{<}\overline{D'}{>}\rangle \in DO$	By sub-derivation of Df-Sigma
$H[\theta] = O = \langle C{<}\overline{D}{>}\rangle \in DO$	Since $\theta \in dom(S)$
$H[\ell] = O_\ell = \langle C_\ell{<}\overline{D_\ell}{>}\rangle \in DO$	Since $\ell \in dom(S)$
$H[v] = O_v = \langle C_v{<}\overline{D_v}{>}\rangle \in DO$	Since $v \in dom(S)$

this proves (1), and (2).

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$	By assumption
$\forall \ell \in dom(S),\ \Sigma[\ell] = C{<}\overline{\ell'.d}{>}$	Since $\Sigma \vdash S$

$H[\theta] = O_C = \langle C{<}\overline{D}{>}\rangle \in DO$
and $\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j)\rangle \in rng(DD)$
and $\forall d_i \in domains(C{<}\overline{\theta'.d}{>})$
$K[\theta.d_i] = D_i = \langle D_{id_i}, C{::}d_i\rangle\ \{(O_C, C{::}d_i) \mapsto D_i\} \in DD$

$\emptyset, \emptyset, G \vdash_O \ell.f_i = v$ By assumption:

Since $e_0 = \ell \in dom(H) \quad e_1 = v$:

$\ell : \Sigma[\ell] = C_\ell\overline{<p>}\ (T'_i\ f_i) \in fields(C_\ell\overline{<p>}) = \overline{T'}\ \overline{f}\ T'_i = C_i\overline{<p'>}$ By sub-derivation of Df-Write

$v : \Sigma[v] = C_v\overline{<p''>} \quad \Sigma[v] <: T'_i$ By sub-derivation of Df-Write

$G \vdash_O export(\Sigma[\ell], \Sigma[v])$ By sub-derivation of Df-Write

$G \vdash_O O_i \in lookup(\Sigma[\ell])$ By sub-derivation of Aux-Export

$G \vdash_O O_j \in lookup(\Sigma[v])$ By sub-derivation of Aux-Export

$\langle O, O_i, O_j, Exp\rangle \in DE$ By sub-derivation of Aux-Export

$\Sigma[\ell] = C_\ell\overline{<\ell'.d>} \quad H[\ell] = O_\ell = \langle C_\ell\overline{<D_\ell>}\rangle \in DO$

$\emptyset, \emptyset, G \vdash_O \ell$ By sub-derivation of Df-Write

$\emptyset; \Sigma; \theta \vdash \ell : \Sigma[\ell]$ By Hypothesis

$G \vdash_{H[\theta]} H[\ell] \in lookup(\Sigma[\ell])$ By Lookup Lemma

$\emptyset, \emptyset, G \vdash_O v$ By sub-derivation of Df-Write

$\emptyset; \Sigma; \theta \vdash v : \Sigma[v]$ By Hypothesis

$\Sigma[v] = C_v\overline{<v'.d>} \quad H[v] = O_v = \langle C_v\overline{<D_v>}\rangle \in DO$

$G \vdash_{H[\theta]} H[v] \in lookup(\Sigma[v])$ By Lookup Lemma

$\Sigma[v] = C_v\overline{<v'.d>} \quad H[v] = O_v = \langle C_v\overline{<D_v>}\rangle \in DO$

$\forall v'_j.d_j \in \overline{v'.d}\ K[v'_j.d_j] = D_j \in rng(DD)$ By Df-Approx

$\Sigma[v] <: T'_i$ By sub-derivation of Df-Write

$H; K; L_I; L_E \vdash H[v] \in irLookup(T'_i)$ By inversion of IR-Lookup

Take $T_k = T'_i \in \overline{T'}$, this proves (3).

Subcase $e_0 = e'_0$. Then $e = (e'_0.f_i = e_1)$

From IRC-Write-Rcv:

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E}{\begin{array}{c}\theta \vdash e_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ e'_0.f_i = e_1; S'; H'; K'; L'_I; L'_E\end{array}}$$

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash e'_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G e''_0.f_i = e_1; S'; H'; K'; L'_I; L'_E$ By IRC-Write-Rcv

Take $e' = (e''_0.f_i = e_1)$.

Subcase $e_0 = v$, and $e_1 = e'_1$. Then $e = (v.f_i = e'_1)$

From IRC-Write-Arg:

$$\frac{\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E}{\begin{array}{c}\theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ v.f_i = e'_1; S'; H'; K'; L'_I; L'_E\end{array}}$$

$\Gamma, \Upsilon, G \vdash_O e_1$ By sub-derivation of DF-WRITE

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'$ By induction hypothesis

$\theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G v.f_i = e_1'; S'; H'; K'; L_I'; L_E'$ By IRC-WRITE-ARG

Take $e' = (v.f_i = e_1')$.

Case DF-INVK : e = $e_0.m(\overline{e})$. There are three subcases to consider, depending on whether the receiver e_0, or the arguments $\overline{e}$ are values.

Subcase $e_0 = \ell$, and $\overline{e} = \overline{v}$ that is $e = \ell.m(\overline{v})$

From IR-INVK:

$$\frac{\begin{array}{c} S[\ell] = C\text{<}\overline{p}\text{>}(\overline{v}) \qquad mbody(m, C\text{<}\overline{p}\text{>}) = (\overline{x}, e_R) \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad mtype(m, C\text{<}\overline{p}\text{>}) = \overline{T} \to T_R \\ H; K; L_I; L_E \vdash O_r \in irLookup(T_R) \qquad E' = \langle O_\ell, O, O_r, Imp\rangle \in DE \qquad L_I' = L_I[(\ell, \theta) \mapsto_\cup \{E'\}] \\ \forall k \in 1..|\overline{x}|\ O_k = H[v_k] \qquad H; K; L_I; L_E \vdash O_k \in irLookup(T_k) \qquad T_k \in \overline{T} \\ E_k = \langle O, O_\ell, O_k, Exp\rangle \in DE \qquad L_E' = L_E[(\theta, \ell) \mapsto_\cup \{E_k\}] \end{array}}{\theta \vdash \ell.m(\overline{v}); S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S; H; K; L_I'; L_E'}$$

To show:

(1) $O = H[\theta]$

(2) $O_\ell = H[\ell]$

(3) $mtype(m, C_\ell\text{<}\overline{p}\text{>}) = \overline{T} \to T_R \quad \forall O_r.H; K; L_I; L_E \vdash O_r \in irLookup(T_R) \quad E' = \langle O_\ell, O, O_r, Imp\rangle \in DE$

(4) $\forall k \in 1..|\overline{T}|\ O_k = H[v_k] \quad H; K; L_I; L_E \vdash O_k \in irLookup(T_k) \quad E_k = \langle O, O_\ell, O_k, Exp\rangle \in DE$

$G \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell' \in dom(S), \Sigma[\ell'] = C' <\overline{p}>$ By sub-derivation of Df-Sigma

$H[\ell'] = O' = \langle C' <\overline{D'}> \rangle \in DO$ By sub-derivation of Df-Sigma

$H[\theta] = O = \langle C <\overline{D}> \rangle \in DO$ Since $\theta \in dom(S)$

$H[\ell] = O_\ell = \langle C_\ell <\overline{D_\ell}> \rangle \in DO$ Since $\ell \in dom(S)$

this proves (1), and (2).

$H[\theta] = O_C = \langle C <\overline{D}> \rangle \in DO$
$and\ \forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, qual(\theta'_j.d_j) \rangle \in rng(DD)$
$and\ \forall d_i \in domains(C <\overline{\theta'.d}>)$

$K[\theta.d_i] = D_i = \langle D_{id_i}, C::d_i \rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$ By Df-Approx

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E'_k \in L_I[(\ell_{src}, \theta)]\ E'_k = \langle H[\ell_{src}], H[\theta], O_k, Imp \rangle \in DE$ By Df-Approx

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$

$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}$

$H; K; L_I; L_E \vdash O_k \in irLookup(T_k)$

$E_k \in L_E[(\theta, \ell_{dst})]\ E_k = \langle H[\theta], H[\ell_{dst}], O_k, Exp \rangle \in DE$ By Df-Approx

$\emptyset, \emptyset, G \vdash_O \ell.m(\overline{v})$	By assumption
$\ell : \Sigma[\ell] = C_\ell\langle\overline{\ell'.d}\rangle$	By sub-derivation of Df-Invk
$mtype(m, C_\ell\langle\overline{\ell'.d}\rangle) = \overline{T'} \to T'_R$	By sub-derivation of Df-Invk
$mtypeDecl(m, C_\ell) = \overline{T_f} \to T_R$	By sub-derivation of Df-Invk
$G \vdash_O import(\Sigma[\ell], T_R)$	By sub-derivation of Df-Invk
$G \vdash_O O_i \in lookup(\Sigma[\ell])$	By subderivation of Aux-Import
$G \vdash_{O_i} O_j \in lookup(T_R)$	By subderivation of Aux-Import
$E' = \langle O_i, O, O_j, Exp\rangle \in DE$	By subderivation of Aux-Import
$\theta \vdash H; K; L_I; L_E, \texttt{new}\ C_\ell\langle\overline{\ell'.d}\rangle(\overline{v}) \rightsquigarrow_G \ell; S'; K'; L'_I; L'_E$	By induction hypothesis
$\emptyset, \Sigma, \theta \vdash \ell : \Sigma[\ell]$	By hypothesis
$\emptyset, \emptyset, G \vdash_O \ell$	By sub-derivation of Df-Invk
$G \vdash_O H[\ell] \in lookup(\Sigma[\ell])$	By Lookup Lemma
$T'_R = \Sigma[\ell''] = C_R\langle\overline{\ell'''.d}\rangle \quad T_R = C_R\langle\overline{p}\rangle$	
$C_R\langle\overline{\ell'''.d}\rangle = [\ldots \ell'''_j.d_j/p_j \ldots]C_R\langle\overline{p}\rangle$	By definition of $mtype, mtypeDecl$
$\forall H; K; L_I; L_E \vdash O_r \in irLookup(T'_R)$	
$G \vdash_{H[\ell]} O_r \in lookup(T_R)$	To Show
$O_r \in rng(H) \quad O_r = \langle C'\langle\overline{D'}\rangle\rangle \quad C' <: C_R$	By subderivation of IR-Lookup
$\forall \ell'''_j.d_j \in \overline{\ell''.d} \quad D''_j = K[\ell'''_j.d_j] \quad D''_j = D'_j$	By subderivation of IR-Lookup

To show

$\forall p_j \in \overline{p} \quad G \vdash_{H[\ell]} D''_j \in findD(C_\ell{::}p_j) \quad D''_j = D'_j = K[\ell'''_j.d_j]$

p_j is a domain parameter, local domain of ℓ or `shared`. Therefore we split the proof in cases

Case domain parameter: $p_j = \alpha_i$

$$\exists \ell'_i.d_i \in \overline{\ell'.d} \text{ s.t } K[\ell'''_j.d_j] = K[\ell'_i.d_i] = D_{\ell i} \quad D_{\ell i} \in \overline{D_\ell} \quad H[\ell] = O_\ell = \langle C_\ell\langle\overline{D_\ell}\rangle\rangle$$

$$D'_j = D_{\ell i} = DD[(H[\ell], C_\ell{::}\alpha_i)] = DD[(H[\ell], qual(\ell'_i.d_i))] = K[\ell'_i.d_i] \qquad \text{By Params Lemma}$$

Case local domain: $p_k = \ell.d_j$

$$D_j = DD[(H[\ell], C_\ell{::}d_j)] = K[\ell.d_j] \qquad \text{By sub-derivation of Df-New}$$

Case local domain: $p_k = \texttt{shared}$

$$D_{vk} = D_{\texttt{shared}} = DD[(H[\ell], {::}\texttt{shared})]$$

We showed

$\forall p_j \in \overline{p} \quad G \vdash_{H[\ell]} D''_j \in findD(C_\ell{::}p_j) \quad D''_j = D'_j$
$G \vdash_{H[\ell]} O_r \in lookup(T_R)$ By inversion of Aux-Lookup
$\forall O_r, E' = \langle H[\ell], H[\theta], O_r, Imp\rangle \in DE$ By above. Take $O_i = H[\ell], O = H[\theta]$

$\forall i \in 1..|\overline{v}|\ v_k : \Sigma[v_k]\ \Sigma[v_k] <: T'_k\ G \vdash_O export(\Sigma[\ell], \Sigma[v_k])$ By sub-derivation of Df-Invk
$\emptyset, \emptyset, G \vdash_O \overline{v}$ By sub-derivation of Df-Invk
$\emptyset, \emptyset, G \vdash_O \ell$ By sub-derivation of Df-Invk
$G \vdash_O H[\ell] \in lookup(\Sigma[\ell])$ By Lookup Lemma
$\forall v_k \in \overline{v} \quad \Sigma[v_k] = C_k{<}\overline{v'.d}{>}$
$\emptyset, \emptyset, G \vdash_O v_k$ By sub-derivation of Df-Invk
$G \vdash_O H[v_k] \in lookup(\Sigma[v_k])$ By Lookup Lemma
$E' = \langle H[\ell], H[\theta], H[v_k], Exp\rangle \in DE$ By subderivation of Aux-Import
$\Sigma[v_k] = C_v{<}\overline{v'.d}{>} \quad H[v_k] = O_v = \langle C_v{<}\overline{D_v}{>}\rangle \in DO$
$\forall v'_j.d_j \in \overline{v'.d}\ K[v'_j.d_j] = D_j \in rng(DD)$ By Df-Approx
$\Sigma[v_k] <: T'_k$ By sub-derivation of Df-Invk
$H; K; L_I; L_E \vdash H[v_k] \in irLookup(T'_k)$ By inversion of IR-Lookup
This proves (4).

Subcase $e_0 = e'_0$ that is $e = e'_0.m(\overline{e})$.
From IRC-RecvInvk:

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E}{\theta \vdash e_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G e'_0.m(\overline{e}); S'; H'; K'; L'_I; L'_E}$$

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis
$\theta \vdash e'_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G e''_0.m(\overline{e}); S'; H'; K'; L'_I; L'_E$ By IRC-RecvInvk
Take $e' = e''_0.m(\overline{e})$.

Subcase $e_0 = v$ that is $e = v.m(v_{1..i-1}, e_i, e_{i+1..n})$.
From IRC-ArgInvk:

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E}{\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G v.m(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E}$$

$\Gamma, \Upsilon, G \vdash_O e_i$ By sub-derivation of Df-Invk
$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By induction hypothesis
$\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G$
$v.m(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E$ By IRC-ArgInvk
Take $e' = v.m(v_{1..i-1}, e'_i, e_{i+1..n})$.

Case DF-CONTEXT : e $= \ell \triangleright e_0$. there are two subcases to consider, depending on whether e_0 is a value

Subcase e_0 is a value that is $e = \ell \triangleright v$.
From IR-CONTEXT:

$$\frac{}{\theta \vdash \ell \triangleright v; S; H; K; L_I; L_E \rightsquigarrow_G v;\ S; H; K; L_I; L_E}$$

Then IR-CONTEXT can apply. Take $e' = v$.

Subcase e_0 is not a value that is $e = \ell \triangleright e_0'$.
From IRC-CONTEXT:

$$\frac{\ell \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L_I'; L_E'}{\theta \vdash \ell \triangleright e; S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright e'; S'; H'; K'; L_I'; L_E'}$$

$\ell \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$	By induction hypothesis
$\theta \vdash \ell \triangleright e_0; S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright e_0'; S'; H'; K'; L_I'; L_E'$	By IRC-CONTEXT
Take $e' = \ell \triangleright e_0'$.	

□

$$\frac{}{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e; S; H; K; L_I; L_E}[\textsc{Df-Reflex}]$$

$$\frac{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e''; S''; H''; K''; L''_I; L''_E \quad \theta \vdash e''; S''; H''; K''; L''_I; L''_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E}{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e'; S'; H'; K'; L'_I; L'_E}[\textsc{Df-Trans}]$$

Figure 9.1: Reflexive, transitive closure of the instrumented evaluation relation

9.3 Theorem: Object Graph Soundness

$$\begin{array}{l} \textit{If} \\ \quad G = \langle DO, DD, DE \rangle \\ \quad DO, DD, DE \vdash (CT, e_{root}) \\ \quad \forall e,\ \theta_0 \vdash e; \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E \\ \quad \Sigma \vdash S \\ \textit{then} \\ \quad DO, DD, DE \vdash_{CT,H} \Sigma \\ \quad (S, H, K, L_I, L_E) \sim (DO, DD, DE) \end{array}$$

where $\rightsquigarrow^*_G$ relation is the reflexive and transitive closure of $\rightsquigarrow_G$ relation (Fig. 9.1). θ_0 is the location of the first object instantiated by e_{root}.

To prove the Object Graph Soundness theorem, we need to show:

(1) $DO, DD, DE \vdash_{CT,H} \Sigma$

(2) $(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

Proof. The proof is by induction on the $\rightsquigarrow^*_G$ relation. There are two cases to consider: [1]

Case Case Df-Reflex :

Since $S = \emptyset$:

$(S, H, K, L_I, L_E) \sim G$

Immediately, from Df-Sigma store constraint with $S = \emptyset$:

$DO, DD, DE \vdash_{CT,H} \Sigma$

Case Case Df-Trans :

By assumption:

$\theta_0 \vdash e; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E$

Since $S = \emptyset$:

$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \sim G$

By inversion of Df-Trans:

$\theta_0 \vdash e; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e'; S'; H'; K'; L'_I; L'_E$

By induction hypothesis:

$(S'; H'; K'; L'_I; L'_E) \sim G$

By inversion of Df-Trans:

$\theta_0 \vdash e'; S'; H'; K'; L'_I; L'_E \rightsquigarrow_G e; S; H; K; L_I; L_E$

[1] The soundness proof follows similar steps to the one of points-to analysis [4].

By preservation:
$(S; H; K; L_I; L_E) \sim G$

By assumption:
$\theta_0 \vdash e; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E$
Since $S = \emptyset$:
$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \sim G$
By inversion of Df-Trans:
$\theta_0 \vdash e; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e'; S'; H'; K'; L'_I; L'_E$
By induction hypothesis:
$DO, DD, DE \vdash_{CT,H'} \Sigma'$
By inversion of Df-Trans:
$\theta_0 \vdash e'; S'; H'; K'; L'_I; L'_E \rightsquigarrow_G e; S; H; K; L_I; L_E$
By preservation:
$DO, DD, DE \vdash_{CT,H} \Sigma$

□

9.4 Lemmas

To prove the Progress and Preservation theorems, we use the following lemmas.

Df-Substitution Lemma.
If
$\Gamma \cup \{\overline{x} : \overline{T_f}\}, \Sigma, \theta \vdash e : T$
$\Gamma \cup \{\overline{x} : \overline{T_f}\}, \Upsilon, G \vdash_O e$
$\Gamma, \Sigma, \theta \vdash \overline{v} : \overline{T_a}$ where $\overline{T_a} <: [\overline{v}/\overline{x}]\overline{T_f}$
then
$\Gamma, \Sigma, \theta \vdash [\overline{v}/\overline{x}]e : T'$ *for some* $T' <: [\overline{v}/\overline{x}]T$
$\Gamma, \Upsilon, G \vdash_O [\overline{v}/\overline{x}]e$

Proof. By induction on the $\Gamma, \Upsilon, G \vdash_O e$ relation. □

Df-Weakening Lemma.
If
$\Gamma, \Upsilon, G \vdash_O e$
then
$\Gamma, \Upsilon \cup \{C\!<\!\overline{D}\!>\}, G, \vdash_O e$

Proof. By induction on the $\Gamma, \Upsilon, G \vdash_O e$ relation. □

Df-Strengthening Lemma.
If
$\Gamma, \emptyset, G \vdash_O \texttt{new}\ C\!<\!\overline{p}\!>\!(v)$
$\forall i \in 1..|\overline{p}| \quad D_i = DD[(O, p_i)]$
$\Gamma, \Upsilon \cup \{C\!<\!\overline{D}\!>\}, G, \vdash_{O'} e'$
then
$\Gamma, \Upsilon, G, \vdash_O e$

Proof. By induction on the $\Gamma, \Upsilon, G \vdash_O e$ relation. □

Df-Domains Lemma.

If

$\emptyset, \Sigma, \theta \vdash e : T$
$\Sigma \vdash S$
$G \vdash_{CT,H} \Sigma$
$\emptyset, \Upsilon, G \vdash_O$ `new` $C\langle\overline{p}\rangle(\overline{v})$
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$
$\Upsilon, G \vdash_O ddomains(C, O_C)$
$\forall i \in 1..|\overline{p}| \quad D_i = DD[(O_C, qual(p_i))]$
$O_C = \langle C\langle\overline{D}\rangle\rangle \quad \{O_C\} \subseteq DO$

then

$\forall d_j \in domains(C\langle\overline{p}\rangle) \quad D_j = DD[(O_C, C::d_j)]$

Proof. By induction on the $\upsilon, G \vdash_O ddomains(C, O_C)$ relation.

Case Aux-Dom :

$\emptyset, \Upsilon, G \vdash_O$ `new` $C\langle\overline{\ell'.d}\rangle(\overline{v})$	By assumption
$CT(C) =$ `class` $C\langle\overline{\alpha}, \overline{\beta}\rangle$ `extends` $C'\langle\overline{\alpha}\rangle$ $\{\ldots\ \overline{dom};\ \ldots;\ \}$	By sub-derivation of Df-New
$G = \langle DO, DD, DE\rangle$	By sub-derivation of Df - New
$\forall i \in 1..\|\overline{\ell'.d}\| \quad G \vdash_O D_i \in findD(C_{this}::\ell'_i.d_i)$	By sub-derivation of Df-New
$O_C = \langle\ C\langle\overline{D}\rangle\ \rangle \quad \{O_C\} \subseteq DO$	By sub-derivation of Df-New
$G \vdash_O dparams(C, O_C)$	By sub-derivation of Df-New
$\{(O_C, C::\alpha_i) \mapsto D_i\} \subseteq DD \quad \alpha_i \in params(C)$	By sub-derivation of AuxAlpha
$\{(O_C, qual(\ell'_i.d_i)) \mapsto D_i\} \subseteq DD$	By sub-derivation of Df-New
$\Upsilon, G \vdash_O ddomains(C, O_C)$	By sub-derivation of Df-New
$O_C = \langle C\langle\overline{D}\rangle\rangle$	By sub-derivation of Df-Dom

Subcase reuse ODomain That is, $\exists O'_C = \langle C''\langle\overline{D''}\rangle\rangle, C''\langle\overline{D''}\rangle \in \Upsilon \cup \{C\langle\overline{D}\rangle\} \implies C'' = C \wedge D_j = DD[(O'_C, C::d_j)]$

$(\texttt{domain}\ d_j) \in \overline{dom} \quad D_j = DD[(O'_C, C::d_j)]$	By sub-derivation of Df-Dom
$\{(O_C, C::d_j) \mapsto D_j\} \subseteq DD$	By sub-derivation of Df-Dom
$\overline{dom} \subseteq domains(C\langle\overline{p}\rangle)$	By definition of *domains*
$\Upsilon, G \vdash_O ddomains(C', O_C)$	By sub-derivation of Df-Dom

Subcase fresh ODomain That is, $\forall C''\langle\overline{D''}\rangle \in \Upsilon \cup \{C\langle\overline{D}\rangle\} \implies C'' \neq C$

$(\texttt{domain}\ d_j) \in \overline{dom} \quad D_j = \langle D_{id_j},\ C::d_j\rangle$	By sub-derivation of Df-Dom
$\{(O_C, C::d_j) \mapsto D_j\} \subseteq DD$	By sub-derivation of Df-Dom
$\overline{dom} \subseteq domains(C\langle\overline{p}\rangle)$	By definition of *domains*
$\Upsilon, G \vdash_O ddomains(C', O_C)$	By sub-derivation of Df-Dom

Subcase $C' \neq \texttt{Object}$

by Induction hypothesis

Subcase $C' = \texttt{Object}$

$domains(\texttt{Object}<\overline{\alpha_0}>) = \emptyset$ by definition of *Aux-Domain-Obj*

Case Aux-Obj1 : Is immediate.

□

Lookup Lemma

If $\emptyset, \Sigma, \theta \vdash \ell : \Sigma[\ell]$
$\Sigma \vdash S$
$G \vdash_{CT,H} \Sigma$
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$
then
$G \vdash_{H[\theta]} H[\ell] \in lookup(\Sigma[\ell])$

Proof.

$\Sigma[\ell] = C<\overline{\ell'.d}>$ By $\Sigma \vdash S$
$H[\ell] = \langle C_\ell<\overline{D}>\rangle$

To Show:

Take $H[\theta] = C_\theta<\overline{D_\theta}>$ $\theta \in domain(H)$
$\forall \ell'_j.d_j \in \overline{\ell'.d}, G \vdash_{H[\theta]} D'_j \in findD(C_\theta :: qual(\ell'_j.d_j))$ $D'_j = D_j$ $D_j \in \overline{D}$ $K[\ell'_j.d_j] = D_j$

Proof by generalized induction.

We first prove two base cases: (1) when d_j is a locally declared domain and (2) when d_j is a locally declared domain of ℓ in the presence of recursive types.

Case $\ell'_j = \theta$. Local domains.

$H[\theta] = \langle C_\theta<\overline{D_\theta}>\rangle \in DO$
$\forall d_j \in domains(\Sigma[\theta])$
$K[\theta.d_j] = D_j = \langle D_{id_j}, C_\theta::d_j\rangle$ $\{(H[\theta], C_\theta::d_j) \mapsto D_j\} \in DD$ By Df-Approx
$D'_j = DD[(H[\theta], C_\theta::d_j)] = K[\theta.d_j]$ By above
$G \vdash_{H[\theta]} D'_j \in findD(C_\theta :: \texttt{this}.d_j)$ By inversion of Aux-FindThis
$G \vdash_{H[\theta]} D'_j \in findD(C_\theta :: qual(\ell'_j.d_j))$ Since$\ell'_j.d_j$
$D'_j = K[\theta.d_j] = K[\ell'_j.d_j] = D_j$ By hypothesis and $\ell'_j = \theta$

Case $\ell'_j = \ell$. Recursive types.

$H[\ell] = O_C = \langle C\overline{<D>}\rangle \in DO$
$and\ \forall \ell'_j.d_j \in \overline{\ell'.d}\ K[\ell'_j.d_j] = D_j = \langle D_{id_j}, qual(\ell'_j.d_j)\rangle \in rng(DD)$
$and\ \forall d_i \in domains(C\overline{<\ell'.d>})$
$K[\ell.d_i] = D_i = \langle D_{id_i}, C::d_i\rangle\ \{(O_C, C::d_i) \mapsto D_i\} \in DD$
$DD[(H[\ell], \ell.d_j)] = D_j = K[\ell.d_j]$ By inversion of Df-New

The induction step.

Case $\ell'_j = n$. d_j is a public domain of n, but not a local domain of θ.

Assume that $\forall \ell' \in domain(H), G \vdash_{H[\theta]} H[\ell'] \in lookup(\Sigma[\ell'])$ but $\ell \notin dom(H)$. Where *domain* means the set of all keys stored in H.

To show:
$K[n.d_j] = D_j \quad G \vdash_{H[\theta]} D_j \in findD(C_\theta::qual(n.d_j))$

$n : \Sigma[n]$
$DD[(H[n], n.d_j)] = D_j = K[n.d_j]$ By Df-Approx, since $d_j \in domains(\Sigma[n])$
$G \vdash_{H[\theta]} H[n] \in lookup(\Sigma[n])$ By induction hypothesis
$G \vdash_{H[\theta]} D_j \in findD(C_\theta::qual(n.d_j))$ By inversion of Aux-Find-Public

$G \vdash_{H[\theta]} H[\ell] \in lookup(\Sigma[\ell])$ By induction

□

Params Lemma.

If $\emptyset, \Sigma, \theta \vdash \ell : \Sigma[\ell]$
$\Sigma \vdash S$
$\emptyset, \emptyset, G \vdash_O$ `new` $C\overline{<\ell'.d>}(\overline{v})$
$G \vdash_{CT,H} \Sigma$
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$
$\ell \in domain(S) \quad H[\ell] = C\overline{<D>} \quad \Sigma[\ell] = C\overline{<\ell'.d>} \quad S[\ell] = C\overline{<\ell'.d>}(\overline{v})$
$CT(C) =$ `class` $C\overline{<\alpha>}$ .... { $\overline{T}\ \overline{f}$; $\overline{dom}$; ...; $\overline{md}$; }
then $\forall \ell'_i.d_i \in \overline{\ell'.d} \quad DD[(H[\ell], qual(\ell'_i.d_i))] = DD[(H[\ell], C::\alpha_i)] = D_i = K[\ell'_i.d_i]$

Proof. By induction on the $G \vdash_O dparams(C, O_C)$ relation using subderivation of Df-New IR-New.

□

BIBLIOGRAPHY

[1] Apache FtpServer 1.0.5. http://mina.apache.org/ftpserver/, 2011.

[2] 42+ Best Practices for Secure Mobile Development. https://viaforensics.com/resources/reports/best-practices-ios-android-secure-mobile-development/, 2013.

[3] ScoriaBench. www.cs.wayne.edu/~mabianto/sb/, 2014.

[4] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, CMU, 2010.

[5] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 321–340, 2009.

[6] M. Abi-Antoun, N. Ammar, and Z. Hailat. Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report. In *QoSA*, 133–142, 2012.

[7] M. Abi-Antoun and J. M. Barnes. Analyzing Security Architectures. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 3–12, 2010.

[8] M. Abi-Antoun, D. Wang, and P. Torr. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security (Short Paper). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 393–396, 2007.

[9] Acme. www.cs.cmu.edu/~acme.

[10] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 1–25, 2004.

[11] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE)*, 187–197, 2002.

[12] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 311–330, 2002.

[13] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented Programming. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 1015–1022, 2009.

[14] K. Ali and O. Lhoták. Application-Only Call Graph Construction. In *European Conference on Object-Oriented Programming (ECOOP)*, 688–712, 2012.

[15] M. Almorsy, J. Grundy, and A. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 100–109, 2012.

[16] M. Almorsy, J. Grundy, and A. Ibrahim. Automated Software Architecture Security Risk Analysis Using Formalized Signatures. In *International Conference on Software Engineering (ICSE)*, 662–671, 2013.

[17] ALTOVA. UModel – UML tool for software modeling and application development. http://www.altova.com/umodel.html, 2013.

[18] N. Ammar and M. Abi-Antoun. Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks. In *Working Conference on Reverse Engineering (WCRE)*, 367–376, 2012.

[19] Android. API Guides. http://developer.android.com/guide/topics/security/, 2012.

[20] S. Arzt, S. Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of Android sources and sinks. Technical Report TUD-CS-2013-0114, TU Darmstadt, 2013.

[21] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *IEEE/ACM International Conference on Automated Software Engineering*

(ASE), 274–277, 2012.

[22] D. Bell. Looking back at the Bell-La Padula model. In *Computer Security Applications Conference*, 336–351, 2005.

[23] E. Bell and L. J. LaPadula. " Secure Computer Systems : Mathematical Foundations. Technical Report MTR-2547, MITRE Corporation, 1973.

[24] B. Berger, K. Sohr, and R. Koschke. Extracting and Analyzing the Implemented Security Architecture of Business Applications. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 285–294, 2013.

[25] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 66–75, 2010.

[26] E. Bodden. FlowDroid Now Supports Implicit Flows. http://sseblog.ec-spride.de/2013/10/flowdroid-implicit-flows/, 2013.

[27] C. Boyapati. *SafeJava: a Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, February 2004.

[28] J. Burns. Mobile Application Security on Android. Black Hat, 2009.

[29] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 441–460, 2007.

[30] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *IEEE International Workshop on Program Comprehension (IWPC)*, 241–247, 2000.

[31] Chess. Improving Computer Security using Extended Static Checking. In *IEEE Symposium on Security and Privacy*, 160–173, 2002.

[32] B. Chess and G. McGraw. Static analysis for security. In *Security & Privacy, IEEE*, 76 – 79, 2004.

[33] Cigital. SecureAssist: Software Security built for Developers. http://www.cigital.com/products/secureassist/, 2013.

[34] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 48–64, 1998.

[35] Coverity. Static Application Security Testing. http://www.coverity.com/security/#SAST, 2013.

[36] J. DelGrosso and G. McGraw. Opinion: Software [in]security – software flaws in application architecture. http://searchsecurity.techtarget.com/opinion/Opinion-Software-insecurity-software-flaws-in-application-architecture, 2013.

[37] W. Dietl, S. Dietzel, M. Ernst, K. Muslu, and T. Schiller. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*, 681–690, 2011.

[38] W. Dietl, S. Drossopoulou, and P. Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(20), 2012.

[39] I. Dillig, T. Dillig, and A. Aiken. Precise Reasoning for Programs Using Containers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 187–200, 2011.

[40] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *Transactions on Software Engineering (TSE)*, 35(4):573–591, 2009.

[41] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[42] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 10, 255–270, 2010.

[43] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application

Security. In *USENIX Conference on Security*, 2011.

[44] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, You, Get Off of My Clipboard - On How Usability Trumps Security in Android Password Managers. In *Financial Cryptography*, 144–161, 2013.

[45] Y. Falcone, S. Currea, and M. Jaber. Runtime Verification and Enforcement for Android Applications with RV-Droid. In *Runtime Verification*, 88–95, 2013.

[46] J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-Grained Access Control with Object-Sensitive Roles. In *European Conference on Object-Oriented Programming (ECOOP)*, 173–194, 2009.

[47] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification (FATES)*, 209–224, 2006.

[48] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. to appear, 2014.

[49] A. Fuchs, A. Chaudhuri, and J. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, Univ. of Maryland, 2009. http://babu.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf.

[50] E. R. Gansner and S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice & Experience*, 30(11):1203–1233, 2000.

[51] Gibson Security. Snapchat - GibSec Full Disclosure. http://gibsonsec.org/snapchat/fulldisclosure/, December 2013.

[52] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[53] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs – A Practical Guide. In *Working Conference on Programming Languages*

(ATPS), Lecture Notes in Informatics (LNI) 215, 123–138, 2013.

[54] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[55] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng. Formally Analyzing Software Architectural Specifications Using SAM. *J. Systems & Software*, 71(1):11–29, 2004.

[56] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Lang. and Comp.*, 13(3):319–339, 2002.

[57] L. Hochstein and M. Lindvall. Diagnosing Architectural Degeneration. In *NASA Goddard Software Engineering Workshop*, 137–142, 2003.

[58] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *ACM SIGPLAN OOPS Messenger*, 3(2):11–16, 1992.

[59] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow Permissions for Android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 652–657, 2013.

[60] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2003.

[61] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[62] HP. Fortify Static Code Analyzer, 2013.

[63] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *ACM SIGMOD International Conference on Management of Data*, 1213–1216, 2011.

[64] W. Huang, W. Dietl, A. Milanova, and M. Ernst. Inference and Checking of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, 1–25, 2012.

[65] IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net, 2012.

[66] IBM. AppScan Source. http://www-03.ibm.com/software/products/en/appscan-

source/, 2013.

[67] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *Transactions on Software Engineering (TSE)*, 27(2):156–169, 2001.

[68] J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich. Static Execute After/Before as a replacement of traditional software dependencies. In *IEEE International Conference on Software Maintenance (ICSM)*, 137–146, 2008.

[69] K. Kenan. *Cryptography in the Database*. Addison-Wesley, 2006. Code: http://kevinkenan.blogs.com/downloads/cryptodb_code.zip.

[70] R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In A. D. Lucia and F. Ferrucci, editors, *International Summer School on Software Engineering*, 140–173, 2008.

[71] R. Küsters, T. Truderung, and J. Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *Computer Security Foundations Symposium (CSF)*, 198–212, 2012.

[72] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[73] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming (ECOOP)*, 275–302, 2003.

[74] A. Lienhard, S. Ducasse, and T. Gîrba. Taking an Object-Centric View on Dynamic Information with Object Flow Analysis. *COMLAN*, 35:63–79, 2009.

[75] Y. Liu. *Practical Static Analysis Framework For Inference Of Security-Related Program Properties*. PhD thesis, Rensselaer Polytechnic Institute, 2010.

[76] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 50–56, 2008.

[77] Y. Liu and A. Milanova. Static Information Flow Analysis with Handling of Implicit

Flows and a Study on Effects of Implicit Flows vs Explicit Flows. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 146–155, 2010.

[78] B. Livshits. Stanford SecuriBench Micro. http://suif.stanford.edu/ livshits/work/securibench-micro/, 2006.

[79] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *USENIX Security Symposium*, 271–286, 2005.

[80] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: a UML-Based Modeling Language for Model-Driven Security. In *Intl. Conference on the Unified Modeling Language*, 426–441, 2002.

[81] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley, 2011.

[82] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying Security Invariants in ExpressOS. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 293–304, 2013.

[83] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting Runtime Heaps for Program Understanding. *Transactions on Software Engineering (TSE)*, 39(6):774–786, 2013.

[84] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 365–383, 2005.

[85] MathWorks. Polyspace. http://www.mathworks.com/products/polyspace/, 2012.

[86] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[87] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions On Software Engineering And Methodology (TOSEM)*, 14(1):1–41, 2005.

[88] R. Minelli and M. Lanza. Software Analytics for Mobile Applications – Insights &

Lessons Learned. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 144–153, 2013.

[89] N. Mitchell, E. Schonberg, and G. Sevitsky. Making Sense of Large Heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, 77–97, 2009.

[90] MITRE Corporation. The Common Weakness Enumeration (CWE) Initiative. http://cwe.mitre.org, 2012.

[91] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation *Transactions on Software Engineering (TSE)*, 27(4):364–380, 2001.

[92] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 228–241, 1999.

[93] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. http://www.cs.cornell.edu/jif/, 2001.

[94] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 158–185, 1998.

[95] ObjectAid. The ObjectAid UML Explorer for Eclipse. www.objectaid.com/.

[96] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.

[97] OMG. Object Constraint Language. http://www.omg.org/spec/OCL/, 2013.

[98] Oracle. Java Platform Standard Edition 6 API Specification. http://docs.oracle.com/javase/6/docs/api/, 2012.

[99] OWASP. Open Web Application Security Project (OWASP). https://www.owasp.org/, 2013.

[100] OWASP. WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2013.

[101] Paladion. AppSec tools for Mobile Enthusiasts. InsecureBank.

http://www.paladion.net/downloadapp.html, 2013.

[102] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, 201–212, 2008.

[103] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *IEEE International Conference on Program Comprehension (ICPC)*, 10–19, 2009.

[104] PLAID Research Group. The Crystal Static Analysis Framework. http://code.google.com/p/crystalsaf, 2009.

[105] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst. Immutability. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, 233–269. 2013.

[106] S. Rawshdeh. A Static Analysis to Extract Dataflow Edges from Object-Oriented Programs with Ownership Domain Annotations. Master's thesis, WSU, 2011. www.cs.wayne.edu/~mabianto/students/11_suhib_ms_thesis.pdf.

[107] S. Rawshdeh and M. Abi-Antoun. A Static Analysis to Extract Dataflow Edges from Object-Oriented Programs with Ownership Domain Annotations. Technical report, WSU, 2011.

[108] A. Raza, G. Vogel, and E. Plödereder. Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering. In *International Conference on Reliable Software Technologies (Ada-Europe)*, 71–82, 2006.

[109] A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 199–215, 2005.

[110] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *International Symposium on Software Reliability Engineering (ISSRE)*, 451–460, 2013.

[111] J. Schäfer and A. Poetzsch-Heffter. A Parameterized Type System for Simple Loose Ownership Domains. *Journal of Object Technology*, 6(5):71–100, 2007.

[112] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *Transactions on Software Engineering (TSE)*, 32(7):454–466, 2006.

[113] Y. Smaragdakis, M. Bravenboer, G. Kastrinis, G. Balatsouras, T. T. Bartolomei, and O. Lhotak. DOOP Framework for Java Pointer Analysis. http://doop.program-analysis.org/about.html.

[114] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 17–30, 2011.

[115] K. Sohr and B. Berger. Idea: Towards Architecture-Centric Security Analysis of Software. In *Symposium on Engineering Secure Software and Systems (ESSOS)*, 70–78, 2010.

[116] A. Spiegel. *Automatic Distribution of Object-Oriented Programs.* PhD thesis, FU Berlin, 2002.

[117] M. Sridharan, S. Chandra, J. Dolby, S. Fink, and E. Yahav. Alias Analysis for Object-Oriented Programs. *Dave Clarke, James Noble, and Tobias Wrigstad (eds.) Aliasing in Object-Oriented Programming*, Springer LNCS(7850):156–232, 2013.

[118] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 59–76, 2005.

[119] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/, 2013.

[120] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Systems & Software*, 44(3), 1999.

[121] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[122] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[123] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[124] P. Torr. Demystifying the Threat-Modeling Process. *IEEE Security and Privacy*, 3(5):66–70, 2005.

[125] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Conference on Fundamental Approaches to Software Engineering (FASE)*, 210–225, 2013.

[126] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 87–97, 2009.

[127] University of Maryland. FindBugs™– Find Bugs in Java Programs. http://findbugs.sourceforge.net/, 2007.

[128] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 13–23, 1999.

[129] R. Vanciu and M. Abi-Antoun. Adding Ownership Domain Annotations and Extracting OOGs from Apache FTP Server. Technical report, Wayne State University, 2011. http://www.cs.wayne.edu/~mabianto/tech_reports/VA11a_TR.pdf.

[130] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code. Technical report, WSU, 2011. www.cs.wayne.edu/~mabianto/tech_reports/VA11_TR.pdf.

[131] R. Vanciu and M. Abi-Antoun. Ownership Object Graphs with Dataflow Edges. In *Working Conference on Reverse Engineering (WCRE)*, 267–276, 2012.

[132] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Objects and other Flow Objects.

In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Informal proceedings, 2013.

[133] R. Vanciu and M. Abi-Antoun. Finding Architectural Flaws Using Constraints. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 334–344, 2013.

[134] R. Vanciu and M. Abi-Antoun. Object Graphs with Ownership Domains: an Empirical Study. *Dave Clarke, James Noble, and Tobias Wrigstad (eds.) Aliasing in Object-Oriented Programming*, Springer LNCS(7850):109–155, 2013.

[135] R. Vanciu and V. Rajlich. Hidden Dependencies in Software Systems. In *IEEE International Conference on Software Maintenance (ICSM)*, 1–10, 2010.

[136] VisualVM. Analyzing a Heap Dump Using Object Query Language (OQL), 2013.

[137] T. Xie, E. Martin, and H. Yuan. Automatic Extraction of Abstract-object-state Machines from Unit-test Executions. In *International Conference on Software Engineering (ICSE)*, 835–838, 2006.

[138] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 95–109, 2012.

ABSTRACT

STATIC EXTRACTION OF DATAFLOW COMMUNICATION FOR SECURITY

by

LAURENŢIU RADU VANCIU

May 2014

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Doctor of Philosophy

The cost of security vulnerabilities in widely-deployed code, such as mobile applications, is high. As a result, many companies are using Architectural Risk Analysis (ARA) to find security vulnerabilities before releasing their applications. The existing analyses are focused on finding local coding bugs such as a hard-coded password, rather than architectural flaws such as bypassing the authentication component. During ARA, to find vulnerabilities that are architectural flaws, security architects use a forest-level view of the runtime architecture instead of reading the code. Unfortunately, such a view is often missing from the documentation or is inconsistent with the code.

This thesis contributes Scoria, a semi-automated approach for finding architectural flaws that uses a static analysis to extract from code with annotations an approximation of the runtime architecture as an abstract object graph with dataflow edges that refer to abstract objects. The annotations express local, modular hints about architectural tiers, logical containment, and strict encapsulation, such that the extracted object graph is hierarchical, which provides architects with both high-level and detailed understanding of the runtime architecture. Moreover, the abstract object graph is sound such that it has unique representatives for all objects and dataflow communication that may exist at runtime. Architects assisted by Scoria can write as machine-checkable constraints various security policies that

are documented only informally. The constraints are in terms of object provenance and indirect communication and can find vulnerabilities missed by constraints that focus only on the presence or the absence of communication, or constraints that track only information flow from sources to sinks.

The evaluation consists of expressing several rules from the CERT Secure Coding Standard for Java for which automated detection was previously unavailable. Scoria is also being used to find information disclosure in open-source Android apps. Based on an existing benchmark, Scoria performs better than commercial and research tools in terms of precision and recall. Scoria is thus making Architectural Risk Analysis, which is today mostly manual and informal, a more rigorous, principled and repeatable activity.

AUTOBIOGRAPHICAL STATEMENT

Laurenţiu Radu Vanciu

Education

- M.S. Computer Science, Wayne State University, Detroit MI, USA, December 2009.
- B.S. Computer Science, Babes Bolyai University, Cluj-Napoca, Romania, July 2006.

Peer review publications

1. Vanciu, R. and Abi-Antoun, M. *Finding Architectural Flaws Using Constraints.* In IEEE/ACM Conference on Automated Software Engineering (ASE), pp. 334–344, 2013.
2. Vanciu R. and Abi-Antoun M. *Finding Architectural Flaws in Android Apps Is Easy.* In ACM Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH). Tool Demonstration. pp. 21–22, 2013.
3. Vanciu R. and Abi-Antoun M. *Extracting Dataflow Objects and other Flow Objects.* In International Workshop on Foundations of Object-Oriented Languages (FOOL), Informal proceedings, 15 pages, 2013.
4. Vanciu, R. and Abi-Antoun, M. *Object Graphs with Ownership Domains: an Empirical Study.* In Springer LNCS vol. 7850 Aliasing in Object-Oriented Programming. Types, Analysis and Verification, pp. 109–155, 2013.
5. Abi-Antoun, M., Vanciu, R. and Ammar, N. *Metrics to Identify Where Object-Oriented Program Comprehension Benefits from the Runtime Structure.* In International Workshop on Emerging Trends in Software Metrics (WETSoM), pp. 42–48, 2013.
6. Vanciu, R. and Abi-Antoun, M. *Ownership Object Graphs with Dataflow Edges.* In Working Conference on Reverse Engineering (WCRE), pp. 267–276, 2012.
7. Vanciu, R. and Rajlich, V., *Hidden Dependencies in Software Systems,* in IEEE International Conference on Software Maintenance (ICSM), pp. 1–10, 2010.
8. Petrenko, M., Rajlich, V., Vanciu, R., *Partial Domain Comprehension in Software Evolution and Maintenance,* in IEEE International Conference on Program Comprehension (ICPC), pp. 13–22, 2008.

Technical reports

- Vanciu R. and Abi-Antoun M. *Adding Ownership Domain Annotations and Extracting OOGs from Apache FTP Server.* SEVERE Technical Report, December 2011.
- Vanciu, R. and Abi-Antoun, M. *Extracting Dataflow Communication from Object-Oriented Code.* SEVERE Technical Report, July 2012.
- Vanciu R. and Abi-Antoun M. *Extracting Dataflow Objects and other Flow Objects.* SEVERE Technical Report, August 2013.

Master thesis

- Vanciu, R. Revealing Hidden Dependencies in Software Systems, Master Thesis – Wayne State University, September 2009.