AN INFERENCE AND CHECKING FRAMEWORK FOR CONTEXT-SENSITIVE PLUGGABLE TYPES

By

Wei Huang

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

Ana Milanova, Thesis Adviser

Christopher D. Carothers, Member

Mukkai Krishnamoorthy, Member

Ondřej Lhoták, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2014
(For Graduation May 2014)

© Copyright 2014
by
Wei Huang
All Rights Reserved

CONTENTS

LIST OF TABLES vii

LIST OF FIGURES ix

ACKNOWLEDGMENT xiv

ABSTRACT xv

1. Introduction 1
   - 1.1 Pluggable Types 2
     - 1.1.1 Example: Reference Immutability Types 3
   - 1.2 Type Inference 3
   - 1.3 Contributions 5
   - 1.4 Thesis Outline 7

2. Inference and Checking Framework 8
   - 2.1 Overview 8
   - 2.2 Unified Typing Rules 10
     - 2.2.1 Framework Parameters 10
     - 2.2.2 Typing Rules 11
     - 2.2.3 Method Overriding 13
     - 2.2.4 Ranking over Typings 14
       - 2.2.4.1 Valid Typing 14
       - 2.2.4.2 Objective Function 14
       - 2.2.4.3 Maximal Typing 15
   - 2.3 Unified Type Inference 16
     - 2.3.1 Set-based Solution 16
       - 2.3.1.1 Set Mapping 16
       - 2.3.1.2 Solving Constraints 17
     - 2.3.2 Properties of the Set-based Solution 18
   - 2.4 Type Checking 20

3. Reference Immutability Types . . . 22
3.1 Type Qualifiers and Subtying Relation . . . 23
3.2 Viewpoint Adaptation . . . 25
3.2.1 Context Sensitivity . . . 25
3.2.2 Viewpoint Adaptation Operation . . . 27
3.2.3 Context of Adaptation . . . 27
3.3 Additional Constraints . . . 28
3.4 Instantiated Typing Rules . . . 28
3.5 Type Inference . . . 30
3.5.1 Initial Mapping . . . 30
3.5.2 Objective Function . . . 31
3.5.3 Maximal Typing . . . 31
3.6 Inference Example . . . 33
3.7 Method Purity Inference . . . 34
4. Ownership Types . . . 40
4.1 Type Qualifiers and Subtying Relation . . . 42
4.2 Viewpoint Adaptation . . . 44
4.2.1 Viewpoint Adaptation Operation . . . 44
4.2.2 Context of Adaptation . . . 45
4.2.3 Additional Constraints . . . 46
4.3 Instantiated Typing Rules . . . 46
4.4 Type Inference . . . 46
4.4.1 Initial Mapping . . . 46
4.4.2 Objective Function . . . 47
4.4.3 Maximal Typing . . . 49
4.5 Inference Example . . . 51
5. Universe Types . . . 52
5.1 Type Qualifiers and Subtyping Relation . . . 52
5.2 Viewpoint Adaptation . . . 54
5.2.1 Viewpoint Adaptation Operation . . . 55
5.2.2 Context of Adaptation . . . 55
5.2.3 Additional Constraints . . . 56

5.3 Instantiated Typing Rules 57

5.4 Type Inference 58

5.4.1 Initial Mapping 58

5.4.2 Objective Function 58

5.4.3 Maximal Typing 58

5.5 Inference Example 60

6. Information Flow Systems 62

6.1 SFlow/Integrity for Java Web Applications 63

6.1.1 Type Qualifiers and Subtying Relation 64

6.1.2 Viewpoint Adaptation 65

6.1.2.1 Context Sensitivity 65

6.1.2.2 Viewpoint Adaptation Operation 66

6.1.2.3 Context of Adaptation 67

6.1.3 Composition with Reference Immutability 67

6.1.4 Instantiated Typing Rules 69

6.1.5 Type Inference 71

6.1.5.1 Initial Mapping 71

6.1.5.2 Objective Function 71

6.1.5.3 Method Summary Constraints 72

6.1.6 Inference Example 75

6.1.7 Web Application-Specific Features 76

6.1.7.1 Reflective Object Creation 77

6.1.7.2 Libraries 78

6.1.7.3 Frameworks 78

6.1.7.4 Mapping Data Structures 80

6.2 SFlow/Confidentiality for Android apps 81

6.2.1 Motivating Example 82

6.2.2 Type Qualifiers and Subtyping Relation 83

6.2.3 Viewpoint Adaptation 85

6.2.4 Additional Constraints 85

6.2.5 Instantiated Typing Rules 86

6.2.6 Inference Example 88

6.2.7 Android-Specific Features 90

6.2.7.1 Libraries 90

6.2.7.2 Multiple Entry Points and Callbacks 91

6.2.7.3 Inter-Component Communication (ICC) 93

7. Empirical Results . . . 97
7.1 ReIm . . . 97
7.1.1 Experimental Setup . . . 99
7.1.2 Inference Result . . . 101
7.1.3 Correctness and Precision Evaluation . . . 102
7.1.4 Purity Inference . . . 105
7.1.4.1 Comparison with JPPA . . . 106
7.1.4.2 Comparison with JPure . . . 108
7.2 Universe Types and Ownership Types . . . 108
7.2.1 Experimental Setup . . . 108
7.2.2 Inference Result of Universe Types . . . 109
7.2.3 Inference Result of Ownership Types . . . 110
7.2.4 Comparing Universe Types vs. Ownership Types . . . 112
7.3 SFlow/Integrity . . . 114
7.3.1 Experimental Setup . . . 114
7.3.2 Inference Result . . . 115
7.4 SFlow/Confidentiality . . . 120
7.4.1 DroidBench . . . 120
7.4.2 Google Play Store . . . 120
7.4.3 Malware . . . 123
8. Related Work . . . 124
8.1 Type Inference and Checking Frameworks . . . 124
8.1.1 Polyglot . . . 124
8.1.2 JastAdd . . . 124
8.1.3 JQual . . . 125
8.1.4 JavaCOP . . . 126
8.1.5 Checker Framework . . . 127
8.2 Reference Immutability Systems . . . 127
8.2.1 Comparison with Javari . . . 128
8.2.2 Comparison with Javarifier . . . 129
8.2.3 Purity . . . 131
8.2.4 Other Related Work on Reference Immutability . . . 132
8.3 Ownership Type Systems . . . 132
8.4 Information Flow Systems . . . 134

8.4.1 Taint Analysis for Web Applications 134
8.4.2 Android Malware Analysis 135
8.5 Other Related Work 137
9. Conclusion and Future Work 138
9.1 Inference and Checking Framework 138
9.2 Instantiations of Pluggable Type Systems 138
9.3 Future Work 139
BIBLIOGRAPHY 140

LIST OF TABLES

4.1 Inference of Ownership Types for the example in Figure 4.1. 51

7.1 Inference results for reference immutability. **#Line** shows the number of lines of the benchmarks, including blank lines and comments. **Annotatable References** include all references, including fields, local variables, return values, formal parameters, and implicit parameters this. It does not include variables of primitive type. **#Ref** is the total number of annotatable references, **#Readonly**, **#Polyread**, and **#Mutable** are the number of references inferred as readonly, polyread, and mutable, respectively. We also include the running time for the benchmarks. The last column **Time** shows the total running time in seconds, including reference immutability inference and type-checking. 102

7.2 Pure methods in Java Olden benchmarks. 106

7.3 The benchmarks used by Ownership Types and Universe Types. 109

7.4 The inference results for Universe Types. Column #Ref gives the total number of references excluding implicit parameters this. Column #Pure gives the number of pure methods inferred automatically based on reference immutability [15]. Columns #any, #rep, and #peer give the number of references inferred as any, rep, and peer, respectively. No user annotations are needed for the inference of Universe Types; therefore, there are only zeros in the #Man column. Last column Time shows the total running time in seconds including parsing the source code, type inference, and type-checking. 109

7.5 The inference results for Ownership Types. Column #Ref again gives the total number of references excluding the implicit parameters this. Columns #⟨rep|_⟩, #⟨own|_⟩, #⟨p|_⟩, and #⟨norep|_⟩ give the numbers of variables whose owners are inferred as rep, own, p, and norep, respectively. The boldfaced number in parentheses in column #⟨rep|_⟩ is an upper bound on rep typings; it is discussed in the text. #Man shows the total number of manual annotations and, in parentheses, the number of annotations per 1kLOC. Time shows the running time in seconds. . . . 110

7.6 Ownership Types vs. Universe Types on allocation sites. The four columns give the number of OT/UT pairings and, in parenthesis, the corresponding percentages. For example, column ⟨rep|_⟩/peer shows the number of allocation sites that were inferred as rep in Ownership Types and peer in Universe Types. 113

7.7 Information about benchmarks and running time of SFlowInfer. The file and line counts include Java files precompiled from JSP files. The time is for running configuration [*Parameter manipulation, SQL injection*]. The time for running other configurations is practically the same. . . . 114

7.8 Inference results for [*Parameter*, *SQL*], [*Parameter*, *XSS*], [*Parameter*, *HTTP*] and [*Parameter*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations; note that a large number of benchmarks have 0 type errors, i.e., they are proven safe. 117

7.9 Inference results for [*Header*, *SQL*], [*Header*, *XSS*], [*Header*, *HTTP*] and [*Header*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations. Again a large number of benchmarks have 0 type errors, i.e., they are proven safe. Due to time constraints, we did not examine the type errors for jspwiki; instead, we conservatively classified them as False-positive. Therefore, the actual False-positive rate is lower than the one reported. 118

7.10 Inference results for [*Cookie*, *SQL*], [*Cookie*, *XSS*], [*Cookie*, *HTTP*] and [*Cookie*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations. Again, we conservatively classified all errors in jspwiki as False-positive and the actual False-positive rate is lower than the one reported. . . . 119

7.11 Summary of comparison on DroidBench [47] with other taint analysis tools ($\surd$ = correct warning, × = false warning, ◯ = missed flow). . . . 120

7.12 Actual flows in Google Play Store shown as Source→Sink pairs. The number in parentheses is the number of type errors reported by DroidInfer for the app. The majority of flows happen in advertising libraries such as InMobi, Millenial Media and Flurry, that are called from the apps. . 121

7.13 Leaks detected in Malware. 123

LIST OF FIGURES

2.1 Inference Framework Architecture. 9

2.2 Syntax. C and D are class names, f is a field name, m is a method name, x, y, z are names of local variables, formal parameters, or parameter this, and q is type qualifier, and i is the call site identifier. As in the code examples, this is explicit. For simplicity, we assume all names are unique. 12

2.3 Unified Typing Rules. Function $typeof$ retrieves the declared qualified types of fields and methods; function $\mathcal{C}$ retrieves the context of adaptation of statement; function $\mathcal{B}$ retrieves the additional constraints imposed by a specific type system; Γ is a type environment that maps variables to qualifiers from U. 13

3.1 Instantiated Typing Rules for ReIm. 29

3.2 Inference Example for ReIm. A.get(Y) has different mutabilities in the contexts of setG and getG. Also, A.getF(), which is called from A.get(Y), has different mutabilities in different calling contexts. The box beside each statement shows the set-based solution; the underlined qualifiers are the final qualifiers picked by ReIm. 33

3.3 A simple linked list and example usage. 35

3.4 Extended typing rules for static fields (see Figure 3.1 for the base type system). Function $methodof(s)$ returns the enclosing method of statement s. Function $statictypeof$(m) returns the static immutability type of m. Static immutability types can be readonly, polyread, or mutable. Rule (TCALL) includes the antecedents from the base type system and the new antecedents that handle the static immutability type of method m, underlined. 36

4.1 Annotated XStack with Ownership Types. The boxed italic letters denote object allocation sites. 43

4.2 Object graph (left) and ownership tree (right) for XStack for the example in Figure 4.1. In the object graph we show all references between objects. In the ownership tree we draw an arrow from the owned object to its owner and put all objects with the same owner into a dashed box. . . . 44

4.3 Instantiated Typing Rules for Ownership Types. For readability we include separate typing rules when the receiver is this. 47

4.4 Ownership trees resulting from typings Γ_1 and Γ_2. Edges $i \rightarrow m$ and $k \rightarrow l$ (shown in red in the object graph) cannot be typed with owner rep simultaneously. 48

5.1 Subtying Hierarchy of Universe Types. 53

5.2 Annotated doubly-linked list. 54

5.3 Object graph (left) and ownership tree (right) for LinkedList. 55

5.4 Instantiated Typing Rules for Universe Types. Again, we show separate typing rules when the receiver is this. 57

5.5 Annotated XStack with Universe Types. The boxed italic letters denote object allocation sites. 60

5.6 Inference of Universe Types for the example in Figure 5.5. 61

6.1 SQL injection example. 63

6.2 SQL injection example with SFlow/Integrity typing. 65

6.3 Context sensitivity example. 66

6.4 Instantiated Typing Rules for SFlow/Integrity. 70

6.5 An example illustrating the computation of the method summary constraints. The frame box beside each statement shows the corresponding constraints the statement generates. 72

6.6 Computation of method summary constraints. C is the set of constraints, it is initialized to the set of constraints for program statements (recall that each equality constraint is written as two subtyping constraints). S is initialized to the result of the set-based solver. Cases 1 and 2 add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C because $q_{\mathsf{y}} \rhd \mathsf{poly}$ always yields q_{y}. Case 3 adds constraints due to transitivity; this case discovers constraints from formals to return values. Case 4 adds constraints between actual(s) and left-hand-side(s) at calls: if there are constraints $q_{\mathsf{w}} <: q_{\mathsf{a}} \rhd q_{\mathsf{x}}$ (flow from actual to formal) and $q_{\mathsf{a}} \rhd q_{\mathsf{y}} <: q_{\mathsf{z}}$ (flow from return value to left-hand-side), and also $q_{\mathsf{x}} <: q_{\mathsf{y}}$ (flow from formal to return value, usually discovered by Case 3), Case 4 adds $q_{\mathsf{w}} <: q_{\mathsf{z}}$. Note that line 4 calls SolveConstraint(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until S stays unchanged. 74

6.7 Aliasing5 example from Stanford SecuriBench Micro. The frame box beside each statement shows the corresponding constraints the statement generates. The oval boxes show propagation during the set-based solution. The constraint at 7 forces url to be tainted, and the constraint at 8 forces b1 to be tainted. The constraint at 3 forces buf to be tainted and the one at 4 forces b2 to be tainted or poly (i.e., the set-based solver removes safe from b2's set). The constraint at 9 then forces str to be tainted or poly. There is a TYPE ERROR at writer.print(str). 75

6.8 The DataSource example due to Ben Livshits [43]. The frame box beside each statement shows the generated constraints correspondingly. The bold red frame boxes show the constraints generated by the algorithm in Figure 6.6. The oval boxes show the set-based solution, where overstruck qualifiers are eliminated by the the algorithm in Figure 6.6. The bold qualifiers are the final maximal typing. It type-checks. 77

6.9 Imprecision caused by mapping data structures. 80

6.10 WallpapersMain leaks the phone identifier (the source at line 16) to a content server (the sink at line 20) in a URL. 83

6.11 Instantiated Typing Rules for SFlow/Confidentiality. 86

6.12 id example. 87

6.13 FieldSensitivity2 example refactored from DroidBench. The frame box beside each statement shows the corresponding constraints the statement generates. We omitted uninteresting constraints. The oval boxes show propagation during the set-based solution. 89

6.14 LocationLeak2 refactored from DroidBench, highlights our inference's novel handling of callback methods. 92

6.15 SMS message stealing in Fakedaum. The SMS message is intercepted in SmsReceiver and passed to TaskService via Intent. Finally, the message is sent out to the Internet using HTTP post method, resulting in a message leak. 95

7.1 Architecture of the implementation of the inference framework. 98

7.2 The elements() method in JOlden/BH 100

7.3 Runtime performance comparison. Note that the running time for type-checking is excluded for both ReImInfer and Javarifier. 103

7.4 Write access to enclosing context results in flatter structure for UT as compared to OT. The bold edge from j to k highlights the write access. 112

7.5 Readonly sharing of internal representation results in flatter structure for OT as compared to UT. The dotted edge from i to e highlights the readonly access. 113

7.6 A PhoneNumber→HttpEntity flow detected by DroidInfer in the textFree app. $\dashrightarrow$ denotes a flow from the left side to the right side. $[_\mathsf{f}$ denotes a write into field f and $]_\mathsf{f}$ denotes a read from field f. The PhoneNumber is encoded in r8, which is written under key 137 in HashMap mPduHeaders, a field of r2. r2 is passed to the constructor of r9, where the mPduHeaders field is retrieved and assigned to field mPduHeader of r9. Later, the value of 137 is retrieved, appended to field mMessage of r9, which eventually is read, converted to a byte array, and sent to the HttpEntity sink. Flows r2 $\dashrightarrow$ r9 and r9 $\dashrightarrow$ r11 have corresponding method summary constraints r2 $<:$ r9 and r9 $<:$ r11; the flows are explained in the red ovals. 122

ACKNOWLEDGMENT

I would like express the deepest appreciation to my advisor Professor Ana Milanova, who has been continuously supporting and helping me of my PhD research. She motivates me by her great ideas and enthusiasm in research, teaches me how to present talks and write papers, and always trusts in me in any circumstances. Without her guidance and persistent help, this thesis would not have been possible.

I also want to thank my thesis committee members for their insightful comments and inspirations.

A special thank to my parents for their endless love and support throughout my life, and to my dearest wife Xihong for her understanding, patience, and encouragement. Finally, I want to thank everyone who has helped and inspired me during my doctoral study.

ABSTRACT

Pluggable types enforce many important program properties. Programmers can use different pluggable type systems to prevent unforeseen runtime errors, facilitate parallelism, program understanding, model checking, and more. With the addition of JSR 308 to Java 8 released in March 2014, pluggable types become part of standard Java.

This thesis presents a framework for specifying, inferring and checking of context-sensitive pluggable types. By supplying a few framework parameters programmers can instantiate the framework's unified typing rules into concrete rules for a specific type system. The framework then takes as input an unannotated or a partially annotated program, infers the most desirable typing (according to the input parameters), and verifies the correctness of the typing. Programmers can use the framework to infer and plug existing type systems, as well as build new type systems.

This thesis presents several instantiations of interesting pluggable type systems: (1) a context-sensitive type system ReIm for reference immutability and an efficient quadratic type inference analysis, (2) the first effective type inference analysis for the classical Ownership Types, (3) a novel quadratic type inference analysis for Universe Types, (4) a context-sensitive type system SFlow/Integrity for detecting information flow vulnerabilities in Java web applications and a novel, worst-case cubic inference analysis, and (5) the dual type system SFlow/Confidentiality for detecting privacy leaks in Android apps and the corresponding inference analysis; the analysis scales well and detects leaks in apps from the Google Play Store and in known malware.

CHAPTER 1
Introduction

Statically typed programming languages like Java and C# incorporate a built-in type system. The system is *mandatory* because every legal program must be type-checked according to the typing rules [1]. Compared to dynamically typed languages such as Perl, Python, Javascript and more, the mandatory typing helps statically typed languages in:

- Providing better documentation in the form of type signatures which can be understood by the compiler;
- Detecting errors during compilation time, e.g. preventing adding an integer to a boolean;
- Giving clearer interface design; and
- Improving performance as many optimizations can be done with type information, e.g. replacing virtual calls by direct calls.

However, there are also disadvantages of the mandatory typing. First, the mandatory typing restricts the expressiveness of statically typed languages. For example, building a prototype with changing requirements using a statically typed language is not as easy using a dynamically typed one, because the types of interfaces of each component may have to be updated as the requirements change.

More importantly, mandatory typing does not always guarantee that "well-typed" programs will not go wrong, because most built-in mandatory type systems do not enforce some important program properties for the languages. For example, programmers have to be very careful not to dereference a null reference, otherwise a RuntimeException will be thrown in Java although the compiler has determined that the program is "well-typed".

1.1 Pluggable Types

As JSR 308 (Type Annotations Specification) [2] is becoming part of Java 8, pluggable types are becoming more and more important. In contrast to mandatory types, pluggable types are optional to programming languages, but they can enforce many different properties. Programmers can use different pluggable type systems to prevent unforeseen runtime errors as well as facilitate parallelism, program understanding, model checking, and more. For example, the Nullness pluggable type system can detect null pointer dereferences, or verify the absence of null pointer dereferences. The Reference immutability pluggable type system can detect unwanted object mutation, or verify the absence of unwanted mutation. With the release of Java 8 in March 2014, programmers can use the standard Java SDK to "plug" informative type annotations and reasoning about their programs.

Pluggable type systems have no effect on the runtime semantics of programming languages [1], which means the existing program would still behave identically at runtime after adding a new pluggable type system. The same code still compiles with any Java compiler and it runs on any JVM. Therefore, different pluggable type systems can be applied on the same program without affecting each other.

In addition, pluggable types can help language evolution [1]. Language designers can first introduce some simple features and gradually add more powerful features by using pluggable types. Because pluggable types do not affect each other, this would guarantee that features introduced by different types do not conflict. What's more, pluggable types make possible unifying dynamic typing and static typing in the same language. For example, Wrigstad et al. present an approach for integrating untyped code and typed code in the same system, then an initial prototype can smoothly evolve into an efficient and robust program [3]. They design a scripting language Thorn which is based on the type system *like types*.

Examples of pluggable type systems include: *nonnull references*, which can detect null pointer dereference errors [4], *reference immutability*, which prevents unwanted object mutations [5, 6], *object ownership*, which enforces object encapsulations and helps modular reasoning about programs [7, 8], *AJ*, which provides data-centric synchronization for Java programs [9, 10], *EnerJ*, which helps improve energy efficiency in scientific computations [11], *confined types*, which prevents internal objects from escaping their protection domain and thus enables writing secure Java programs [12], *scoped types*, which statically enforces programming discipline that eschews complexity and run-time exceptions in favor of simplicity and safety [13], *interning types* for preventing the comparison between interned value and non-interned value [14], and more.

1.1.1 Example: Reference Immutability Types

In this section, we briefly discuss one pluggable type system: reference immutability. An *immutable*, also called readonly reference, cannot be used to modify the state of the object it references, including the transitively reachable state. Consider the following code snippet:

```
1  Matrix transpose(readonly Matrix matrix) {
2      matrix.rowSize = 100; // Compile-time error!
3      ...
4  }
```

where the transpose method is not supposed to mutate matrix. By declaring the parameter matrix as readonly, the reference immutability types prevent the unwanted mutation at line 2.

There are several type systems for reference immutability. In this thesis, we develop ReIm [15], a context-sensitive type system for reference immutability. There are three source-level type qualifiers in ReIm: mutable, readonly, and polyread. A mutable reference can be used to mutate the referenced object, while a readonly reference cannot. And polyread encodes context-sensitivity, which we will discuss in detail in Chapter 3.

1.2 Type Inference

Just as with a traditional type system, a pluggable type system requires annotations in the source code, and most pluggable type systems require a sizable amount of annotations. The annotation burden on programmers may inhibit practical adoption. Programmers would have to manually annotate all variables, while in most cases they would want to annotate only a few variables they care about. In the above transpose example, programmers would annotate the parameter matrix as readonly, because they would want to ensure that there is no mutation through matrix, but they would like to avoid annotating the rest of the local variables, fields and return types.

Therefore, it is important to develop type inference techniques in order to further advance pluggable types. Type inference transforms unannotated or partially-annotated programs into fully-annotated ones. Programmers can annotate a few variables they care about and have the inference infer the rest. Given the release of Java 8, more programmers will be "plugging" informative type annotations to improve code quality, and type inference will be increasingly important.

- First, type inference reduces the annotation burden on programmers and therefore helps the adoption of pluggable types in practice.
- Second, it reveals valuable information about how existing programs use the concepts expressed in the type system.
- Last but not least, it enables easy prototyping of novel type systems. Type system designers can quickly see how their new type system works on existing programs with the help of type inference.

However, type inference is difficult. Most existing type inference algorithms are domain-specific. For example, Quinonez et al. use *guarded constraints* for inferring Reference Immutability [16] and Dietl et al. encode Universe Types' constraints as a boolean satisfiability problem, which is solved by a Max-SAT solver [17]. There are no algorithms for Ownership Types [7], AJ [9], EnerJ [11] and many other useful type systems (to the best of our knowledge). The above mentioned techniques are usually efficient and precise. However, it is not easy to apply these techniques to other new type systems because they are built to target these specific type systems.

In addition, most pluggable type systems permit many different typings for a given program. For example in reference immutability types, leaving every type as mutable is a legal typing, but a useless one that expresses no design intent and detects no coding errors. The fact that multiple valid typings are allowed, is one major source of difficulty in inferring types automatically.

In this thesis, we propose an unified framework with which a large class of pluggable type systems can be *specified*, *inferred*, and *checked*. Programmers supply a few parameters (to be explained in Chapter 2), which instantiate the framework's typing rules into concrete typing rules for a specific system. The framework then takes an unannotated or a partially annotated program and fills in the remaining types. Programmers can use the framework to specify an existing type system and detect/prevent errors in their programs. We have instantiated Ownership Types [7], Universe Types [8], AJ Types [9], and more. Meanwhile, type system designers can use the framework to define type qualifiers and typing rules, and use type inference to quickly evaluate the new type system. We have built several new useful type systems, including ReIm (see Chapter 3), SFlow/Integrity and SFlow/Confidentiality (see Chapter 6), which enable effective information flow analysis for Java Web applications and Android.

Our inference and checking framework has achieved results ranging from mostly theoretical to keenly practical. The instantiation for Ownership Types is the first effective inference analysis for Ownership Types [7] (ownership type inference, a longstanding problem in the ownership community, was the problem we had set to solve originally). The instantiation for ReIm achieves better scalability and equal precision compared to the state-of-the-art reference immutability inference tool, Javarifier [16]. The instantiation for SFlow/Confidentiality is an effective taint analysis, which uncovers privacy leaks in Android apps from the Google Play Store.

1.3 Contributions

The work in this thesis makes the following contributions.

- A unified inference and checking framework for context-sensitive pluggable type systems. By giving five framework parameters, programmers can instantiate the framework's unified typing rules into concrete rules for a specific type system. The framework then infers the most desirable typing and verifies the correctness of the typing for an unannotated or partially annotated program. We have instantiated the framework with several type systems from the literature, including the classical Ownership Types [7], Universe Types [8] and

AJ [9], as well as with several novel type systems, ReIm, SFlow/Integrity and SFlow/Confidentiality.

- A context-sensitive type system ReIm for reference immutability and an efficient type inference analysis with $O(n^2)$ time complexity. In addition, we present a novel application of ReIm, method purity inference. The implementation is evaluated on programs of up to 348kLOC, including widely used Java applications and libraries, comprising 766kLOC in total.

- The first effective type inference analysis for the classical Ownership Types. The type inference is able to type large programs with low annotation burden — on average, 6 annotations per 1kLOC. As far as we know, this is the first type inference that can scale to programs of up to 110kLOC.

- A novel type inference analysis for Universe Types. The analysis has $O(n^2)$ time complexity and requires no manual annotations. The type inference is able to produce the "best typing". In addition, we present a comparison of the two different encapsulation disciplines in ownership types, *owner-as-dominator* and *owner-as-modifier*, by comparing the classical Ownership Types, which enforces the owner-as-dominator discipline, to Universe Types, which enforces the owner-as-modifier discipline.

- A context-sensitive integrity type system SFlow/Integrity for detecting information flow vulnerabilities in Java web applications and a novel, worst-case cubic inference analysis. We present techniques for effective handling of reflective object creation, libraries and frameworks, which are ubiquitous in Java web applications. The empirical evaluation on Java web applications of up to 126kLOC shows the inference analysis is both scalable and precise.

- The dual confidentiality type system SFlow/Confidentiality for preventing privacy leaks in Android apps. It handles effectively the Android-specific features, including "open" programs with multiple entry points, callbacks, large libraries, and inter-component communication. The analysis is evaluated on different sets of Android apps, including micro benchmarks, apps from the

Google Play Store, and known malware. It scales well and reveals numerous leaks. The analysis is one of the first effective tools for Android.

1.4 Thesis Outline

The rest of the thesis is organized follows. Chapter 2 presents the unified inference framework, including the unified typing rules and unified type inference. Then we present several instantiations of the inference framework. Chapter 3 discusses the reference immutability type system ReIm. Chapter 4 and Chapter 5 presents the instantiations for the classical Ownership Types and for Universe Types, respectively. Chapter 6 instantiates the inference framework into information flow systems, presenting two novel applications using type inference: detecting security violations in Java web applications and detecting privacy leaks in Android apps. Chapter 8 discusses the related work and Chapter 9 concludes the thesis and discusses the future work.

CHAPTER 2
Inference and Checking Framework

In this chapter, we present an unified inference framework for context-sensitive pluggable type systems. We observe that a large number of context-sensitive pluggable type systems share some common properties, e.g. they enforce subtyping constraints on explicit or implicit assignments, account for context-sensitivity on field accesses and method calls, and so forth. Therefore, such type systems can be *specified*, *inferred*, and *checked* within an unified framework. By supplying a few parameters (to be explained shortly), programmers can instantiate the framework's unified typing rules into concrete ones for a specific type system. The framework then takes as input an unannotated or a partially annotated program, infers the most desirable typing (according to the input parameters), and verifies the correctness of the typing. As a result, programmers can use the framework to infer and plug existing type systems, as well as build new type systems. In addition, they can quickly (1) see how the concepts expressed in their systems appear in existing programs, and (2) assess the annotation burden of their systems (if any).

The inference framework is an extension of the Checker Framework [18, 14], on which programmers can build and test their type systems. Hence, corresponding type checkers can be built in the Checker Framework to verify the inference results.

In the following sections of this chapter, we present the unified inference framework, including the architecture (Section 2.1), the unified typing rules (Section 2.2), the unified type inference (Section 2.3), and the type-checking (Section 2.4).

2.1 Overview

The architecture of the inference framework is shown in Figure 2.1. It consists of four steps shown in dark grey boxes in the figure. The first step is the *unified typing rules*, which can be instantiated to concrete typing rules by giving five parameters:

Portions of this chapter previously appeared as: W. Huang *et al.*, "Inference and checking of object ownership," in *Proc. European Conf. Object-Oriented Programming*, Beijing, China, 2012, pp. 181–206.

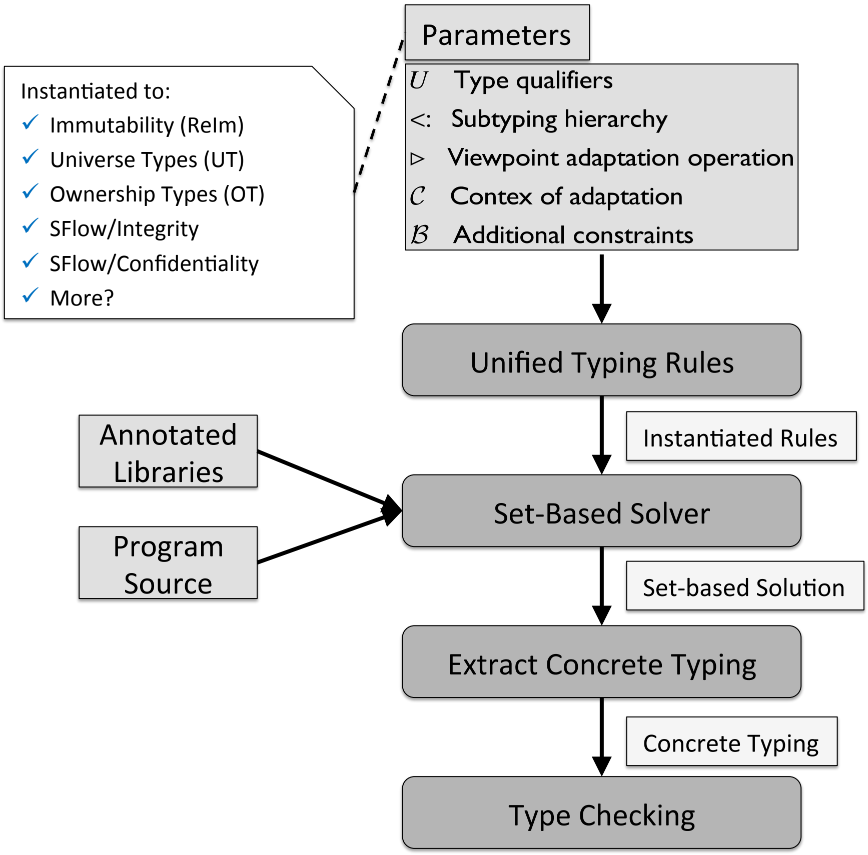


Figure 2.1: Inference Framework Architecture.

the type qualifiers, the subtyping relation, the viewpoint operation, the context of viewpoint adaptation, and the additional constraints enforced by a specific type system. The second step is the *set-based solver*, which takes as input the instantiated typing rules, the program source, and the annotated libraries if necessary, then outputs the *set-based solution* or type errors indicating no valid typing exists. The set-based solution maps variables to the sets of possible type qualifiers and it contains all valid typings. The third step is to extract a concrete typing from the set-based solution. The last step verifies the correctness of the extracted concrete typing according to the instantiated typing rules.

2.2 Unified Typing Rules

In this section, we first describe the five framework parameters, then we explain the unified typing rules.

2.2.1 Framework Parameters

There are five framework parameters for instantiating the unified typing rules: (1) type qualifiers, (2) subtyping relation, (3) viewpoint adaptation rules, (4) context of adaptation, and (5) additional constraints.

Type qualifiers U is the universal set of qualifiers for a specific type system. For example in ReIm [15] (discussed in detail in Chapter 3), $U = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$.

Subtying relation $<:$ defines the subtying relation between type qualifiers. For example, the subtying relation in ReIm is

$$\mathsf{mutable} <: \mathsf{polyread} <: \mathsf{readonly}$$

which means we can assign a mutable or polyread reference to a readonly one, but we cannot assign an readonly reference to a polyread or mutable one. Notice that $<:$ is reflexive and transitive. Therefore, a qualifier is always a subtype of itself ($q <: q$).

Viewpoint adaptation $\rhd$ is a concept from Universe Types [8, 19, 20], which can be adapted to Ownership Types [7] and ownership-like type systems such as AJ [9]. Viewpoint adaptation of a type q' from the point of view of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$. Traditional viewpoint adaptation from Universe Types defines one viewpoint adaptation operation $\rhd$; it uses $\rhd$ to adapt fields, formal parameters, and method returns from the point of view of the *receiver* at the field access or method call. From now on, we refer to q in $q \rhd q'$ as the *context of adaptation*, or simply as the *context*.

The inference framework uses viewpoint adaptation to encode context sensitivity. For example, the type of x.f is not just the declared type of field f — it is the type of f adapted from the point of view of x. As a concrete example, in ReIm, if the declared type of f is $\mathsf{polyread}$, which denotes the mutability of f depends on the enclosing context, then the ReIm type of $\mathsf{x.f}$ is not necessarily $\mathsf{polyread}$, but is $\mathsf{readonly}$ when x is $\mathsf{readonly}$ and $\mathsf{mutable}$ when x is $\mathsf{mutable}$.

Context of adaptation $\mathcal{C}$ is the point of view of the adaptation, i.e. it is q in $q \rhd q'$. Traditional viewpoint adaptation as defined in [8, 20] always takes the *receiver* as the viewpoint at field access or method call.

We generalize traditional viewpoint adaptation. Specifically, we allow for adaptation from *different contexts*, not only from the viewpoint of the receiver. Viewpoint adaptation encodes context sensitivity directly in the typing rules. Varying the viewpoint adaptation operation and/or the choice of viewpoint context at method calls, allows for encoding different kinds of context sensitivity (i.e., different kinds of approximation).

The context of adaptation is defined as a function $\mathcal{C}$ of statement s. $\mathcal{C}(\mathsf{s})$ returns the type of the context of adaptation for statement s. For example, in ReIm, the function for field access $\mathcal{C}(\mathsf{x} = \mathsf{y.f})$ returns the type of y, which is the receiver of the field access. And the function for method call $\mathcal{C}(\mathsf{x} = \mathsf{y.m(z)})$ returns the type of x, which is the left-hand-side of the call assignment.

Additional Constraints $\mathcal{B}$ are used to enforce additional constraints by a specific type system beyond the standard subtying constraints. It is defined as a function $\mathcal{B}$ of statement s. For example, ReIm requires that the receiver must be $\mathsf{mutable}$ when there is a field write. This can be expressed using the additional constraints as $\mathcal{B}(\mathsf{y.f} = \mathsf{x})$, which returns the singleton set of constraints $\{\mathsf{y} = \mathsf{mutable}\}$ (for brevity, for the rest of the paper, we typically use only the variable, e.g., y, instead of the more verbose q_{y}).

2.2.2 Typing Rules

For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [9] whose syntax appears in Figure 2.2. The language models Java with a syntax in a “named form”, where the results of field accesses, method calls, and instantiations are immediately stored in a variable. Without loss of generality, we

$$\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} & \textit{class} \\
fd & ::= t\ \mathsf{f} & \textit{field} \\
md & ::= t\ \mathsf{m}(t\ \mathsf{this},\ t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}}\ \mathsf{s};\ \mathsf{return\ y}\ \} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s};\mathsf{s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} & \textit{statement} \\
 & \mid \mathsf{x} = \mathsf{y.f} \mid \mathsf{y.f} = \mathsf{x} \mid \mathsf{x} = \mathsf{y.m}^{i}(\mathsf{z}) & \\
t & ::= q\ \mathsf{C} & \textit{qualified type}
\end{array}$$

Figure 2.2: Syntax. C and D are class names, f is a field name, m is a method name, x, y, z are names of local variables, formal parameters, or parameter this, and q is type qualifier, and i is the call site identifier. As in the code examples, this is explicit. For simplicity, we assume all names are unique.

assume that methods have parameter this, and exactly one other formal parameter. Features not strictly necessary are omitted from the formalism, but they are handled correctly in the implementation. We write $\overline{t\ \mathsf{y}}$ for a sequence of local variable declarations.

In contrast to a formalization of standard Java, a type t has two orthogonal components: type qualifier q and Java class type C. The type system is *orthogonal* to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

Figure 2.3 shows the unified typing rules over the syntax defined in Figure 2.2. Rule (TNEW) ensures that the instantiated type q is a subtype of the type q_{x} of the left-hand side. It also enforces the additional constraints $\mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C})$. Similarly, rule (TASSIGN) ensures the subtyping relation in assignments. Rule (TWRITE) first invokes $\mathcal{C}(\mathsf{y.f} = \mathsf{x})$ to get the context of adaptation q_c, then makes sure that the type q_{x} of the right-hand-side is a subtype of the adapted type of the field f, namely $q_c \rhd q_{\mathsf{f}}$. Auxiliary function $\mathit{typeof}(\mathsf{f})$ retrieves the type of field f from its declaration. Similarly, rule (TREAD) ensures that the adapted field type is a subtype of the type of the left-hand-side. Rule (TCALL) uses $\mathit{typeof}(\mathsf{m})$ to retrieve the type of method m, namely $q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}}$, where q_{this} the type of the implicit parameter this, q_p is the parameter type and q_{ret} is the return type. Then it creates the subtyping constraint between the type of the receiver and the adapted type of this, between the type of the actual argument z and the adapted type of the formal parameter p, and between the adapted return type and the type of the left-hand-side x of the call assignment.

$$\textsc{(tnew)}\quad \frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}$$

$$\textsc{(tassign)}\quad \frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

$$\textsc{(twrite)}\quad \frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ q_c = \mathcal{C}(\mathsf{y.f} = \mathsf{x}) \quad q_{\mathsf{x}} <: q_c \rhd q_{\mathsf{f}} \\ \mathcal{B}(\mathsf{y.f} = \mathsf{x})\end{array}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

$$\textsc{(tread)}\quad \frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_c = \mathcal{C}(\mathsf{x} = \mathsf{y.f}) \quad q_c \rhd q_{\mathsf{f}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.f})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

$$\textsc{(tcall)}\quad \frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_c = \mathcal{C}(\mathsf{x} = \mathsf{y.m(z)}) \quad q_{\mathsf{y}} <: q_c \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} <: q_c \rhd q_p \quad q_c \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z}))\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}$$

Figure 2.3: Unified Typing Rules. Function *typeof* retrieves the declared qualified types of fields and methods; function $\mathcal{C}$ retrieves the context of adaptation of statement; function $\mathcal{B}$ retrieves the additional constraints imposed by a specific type system; Γ is a type environment that maps variables to qualifiers from U.

The unified typing rules are parameterized in that they can be instantiated into concrete typing rules by giving five framework parameters: type qualifiers U, subtyping relation $<:$, viewpoint adaptation rules $\rhd$, context of adaptation $\mathcal{C}$, and additional constraints $\mathcal{B}$.

2.2.3 Method Overriding

Method overriding is handled by the standard constraints for function subtyping. If m' overrides m we require

$$\mathit{typeof}(\mathsf{m}') <: \mathit{typeof}(\mathsf{m})$$

and thus,

$$(q_{\mathsf{this}_{\mathsf{m}'}}, q_{p_{\mathsf{m}'}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}'}}) \;<:\; (q_{\mathsf{this}_{\mathsf{m}}}, q_{p_{\mathsf{m}}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}}})$$

This entails $q_{\mathsf{this}_{\mathsf{m}}} <: q_{\mathsf{this}_{\mathsf{m}'}}$, $q_{p_{\mathsf{m}}} <: q_{p_{\mathsf{m}'}}$, and $q_{\mathsf{ret}_{\mathsf{m}'}} <: q_{\mathsf{ret}_{\mathsf{m}}}$.

2.2.4 Ranking over Typings

Most pluggable type systems allow many different typings for a given program. For example, ReIm allows many different typings. The trivial typings that apply to every program (in ReIm the trivial typing types all variables mutable) do not express programmers' intent nor detect/prevent coding errors. Our goal is to pick up the most desirable, or best, typing among all permitted typings. This section formalizes the notion of the best typing using *a ranking over all typings.*

2.2.4.1 Valid Typing

We begin by defining the notion of a valid typing. Let P be a program and F be a pluggable type system with universal set of qualifiers U_F. A typing $\Gamma_{P,F}$ is a mapping from variables in P to the type qualifiers in U_F. A typing $\Gamma_{P,F}$ is a valid typing for P in F when it renders P well-typed in F. Note that a valid typing $\Gamma_{P,F}$ must maintain programmer-provided qualifiers in P, that is, if a variable x is annotated by the programmer with q, then for every valid typing $\Gamma_{P,F}$, we have $\Gamma_{P,F}(\mathsf{x}) = q$.

2.2.4.2 Objective Function

We proceed to define an objective function o that can be used to rank valid typings. The objective function o takes a valid typing Γ and returns a tuple of numbers[1]. The tuples are ordered lexicographically.

To create the tuple, the objective function o assumes that the qualifiers are partitioned and the partitions are ordered. Then, each element of the tuple is the number of variables in Γ whose type is in the corresponding partition. For example

[1] Strictly, o and Γ are defined in terms of a specific type system F and program P; for brevity, we omit the subscripts when they are clear from context.

in ReIm, the function is instantiated as

$$o_{ReIm}(\Gamma) = (|\Gamma^{-1}(\textsf{readonly})|, |\Gamma^{-1}(\textsf{polyread})|, |\Gamma^{-1}(\textsf{mutable})|)$$

The partitioning and ordering is

$$\{\textsf{readonly}\} > \{\textsf{polyread}\} > \{\textsf{mutable}\}$$

Each qualifier falls in its own partition. This means, informally, that we prefer readonly over polyread and mutable, and polyread over mutable. More formally, the partitioning and ordering gives rise to a preference ranking O_{ReIm} over all qualifiers:

$$O_{ReIm} : \textsf{readonly} > \textsf{polyread} > \textsf{mutable}$$

Note that this preference ranking is not related to subtyping. Given two typings Γ_1 and Γ_2, we have $\Gamma_1 > \Gamma_2$ iff Γ_1 has a larger number of variables typed readonly than Γ_2, or Γ_1 and Γ_2 have the same number of readonly variables, but Γ_1 has a larger number of polyread variables than Γ_2. Function o_{ReIm} gives a natural ranking over the set of valid typings for ReIm. In fact, the maximal (i.e., best) typing according to the above ranking, *maximizes the number of variables typed* readonly.

The objective function can be instantiated to many other type systems, as we will explain in detail in the following chapters.

2.2.4.3 Maximal Typing

A *maximal typing* is a typing that maximizes o (i.e., the best typing(s) according to the heuristics encoded in o).

Definition 1. Maximal Typing. Given an objective function o over the set of valid typings, a valid typing Γ is a *maximal typing of P in F* under o, if for every valid typing Γ', we have $\Gamma' \neq \Gamma \Rightarrow \Gamma \geq \Gamma'$.

Perhaps somewhat unexpectedly, for ReIm, as well as other interesting systems such as Universe Types [8] and AJ [9], there exists a *unique maximal typing.* For other type systems such as Ownership Types [7], however, in general, there are multiple maximal typings, i.e., there are multiple typings that maximize the objective function. This is discussed in detail in the following chapters.

2.3 Unified Type Inference

The unified inference works on completely unannotated programs, as well as on partially-annotated programs. We believe that neither fully automatic inference nor fully manually annotated programs are feasible choices. In many interesting systems, fully automatic inference is impossible; that is, the programmer must provide initial annotations which typically reflect semantics that is impossible to infer. We envision a cooperative system that fills in as many annotations as possible and queries the programmer for a small set of annotations on certain variables to resolve ambiguities. The system seamlessly integrates programmer-provided annotations with inferred annotations.

The key idea in our system is to compute a set-based solution S instead of a single typing by using a set-based solver. S maps variables to *sets* of qualifiers: for every statement s, for every variable v in s, and for every qualifier $q \in S(v)$, there are qualifiers in the sets of the remaining variables in s, such that q and those qualifiers make statement s type-check.

2.3.1 Set-based Solution

2.3.1.1 Set Mapping

S maps each program variable (annotatable reference) to a *set* of possible type qualifiers. The variables in the mapping can be (1) local variables, (2) parameters (including this), (3) fields, and (4) method returns. For instance, $S(\mathsf{x}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ means the type of reference x can be either readonly or polyread, but not mutable in ReIm.

The initial mapping, S_0, is defined as follows. Programmer-annotated variables are initialized to the singleton set which contains only the programmer-provided annotation. Variables that are not annotated are initialized to the *maximal set* of qualifiers U.

2.3.1.2 Solving Constraints

The inference then creates constraints for all program statements according to the instantiated typing rules in Figure 2.3. Subsequently, the inference iterates over these constraints, and runs SolveConstraint(c) for each constraint c. SolveConstraint(c) takes as input a mapping S and outputs an updated mapping S'. Informally, SolveConstraint(c) removes *infeasible qualifiers* from the set of variables that participate in c. Infeasible qualifiers are those make the constraint unsatisfiable. SolveConstraint(c) refines the set of each variable that participates in c as follows. Let $\mathsf{x}, \mathsf{y}, \mathsf{z}$ be the variables in c. For each variable, SolveConstraint(c) removes all infeasible qualifiers from the variable's set. Consider variable x. SolveConstraint(c) removes each q_{x} from $S(\mathsf{x})$, if there does not exist a pair $q_{\mathsf{y}} \in S(\mathsf{y})$, $q_{\mathsf{z}} \in S(\mathsf{z})$ such that $q_{\mathsf{x}}, q_{\mathsf{y}}, q_{\mathsf{z}}$ satisfy c. The same repeats for y, and then z. Note that the order in which references are examined does not affect the final result — one can see that SolveConstraint(c) always removes the same set of qualifiers from $S(\mathsf{x})$, regardless of whether x is examined first, second or last.

More formally, SolveConstraint(c) is defined as follows:

```
foreach $v_i \in c$
  $S'(v_i) = \{\ q_i \mid q_i \in S(v_i)$ and
      $\exists q_1 \in S(v_1), \ldots, q_{i-1} \in S(v_{i-1}), q_{i+1} \in S(v_{i+1}), \ldots, q_k \in S(v_k)$
          s.t. $q_1, \ldots, q_k$ satisfy the subtying, viewpoint adaptation, and $\mathcal{B}$ in $c\ \}$
```

After applying SolveConstraint(c), for each variable $v_i \in c$ and each $q_i \in S'(v_i)$, there exist $q_1 \in S'(v_1)$, ..., $q_{i-1} \in S'(v_{i-1})$, $q_{i+1} \in S'(v_{i+1})$, ..., $q_k \in S'(v_k)$, such that q_1, ..., q_k satisfy the constraint c.

For example, consider statement $\mathsf{x} = \mathsf{y}$ for ReIm. The inference generates the corresponding constraint $c : \mathsf{y} <: \mathsf{x}$. Suppose that $S(\mathsf{x}) = \{\mathsf{mutable}\}$ and $S(\mathsf{y}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$ before SolveConstraint(c). The function removes $\mathsf{readonly}$ and $\mathsf{polyread}$ from $S(\mathsf{y})$ because there does not exist $q_{\mathsf{x}} \in S(\mathsf{x})$ that satisfies $\mathsf{readonly}\ <:\ q_{\mathsf{x}}$ or $\mathsf{polyread}\ <:\ q_{\mathsf{x}}$. After the application of the function, S' is as follows: $S'(\mathsf{x}) = \{\mathsf{mutable}\}$ and $S'(\mathsf{y}) = \{\mathsf{mutable}\}$. Here, $\mathsf{readonly}$ and $\mathsf{polyread}$ are infeasible qualifiers. In the case that the infeasible qualifier is the last element in $S(\mathsf{x})$, the inference keeps this qualifier in $S(\mathsf{x})$ and reports a *type error* at c. For example, if $S(\mathsf{x}) = \{\mathsf{mutable}\}$ and $S(\mathsf{y}) = \{\mathsf{readonly}\}$, then the solver reports a type error on constraint $\mathsf{y} <: \mathsf{x}$ because it is not satisfiable. We keep the qualifier in order to produce better error reports: a type error $\mathsf{y}\{\mathsf{readonly}\} <: \mathsf{x}\{\mathsf{mutable}\}$ is more informative than $\mathsf{y}\{\mathsf{readonly}\} <: \mathsf{x}\{\}$.

The inference is a fixpoint analysis. It keeps removing infeasible qualifiers for each constraint until S reaches the fixpoint, i.e. S remains unchanged from the previous iteration. There are two outcomes: (1) No type errors in which case the inference outputs a set-based solution, or (2) There are type errors indicating that there are unsatisfiable constraints, which means the program is untypable given the initial programmer-provided annotations.

The computation fits the requirements of a monotone framework [21]. The property space is the standard lattice of subsets, with the set of qualifiers U_F being the bottom 0, and the empty set $\emptyset$ being the top 1 of the lattice. The transfer functions are monotone. Therefore, the set-based solution S produced by fixpoint iteration is the unique least solution (for historical reasons sometimes this solution is referred to as the "maximal fixpoint solution" [21]).

The fixpoint will be reached in $O(n^2)$ time where n is the size of the program. In each iteration, at least one of the $O(n)$ variables is updated to point to a smaller set. Hence, there are at most $O(|U_F|n)$ iterations where $|U_F|$ is the number of elements in the qualifier set U_F which is a small constant, resulting in the $O(n^2)$ time complexity.

2.3.2 Properties of the Set-based Solution

Let us now consider the properties of the set-based solution S. These properties help establish that for certain type systems one can derive a maximal (i.e., best) typing from the set-based solution S.

The first proposition states that if the algorithm removes a qualifier q from the set $S(v)$ for variable v, then there does not exist a valid typing that maps v to q. The notation $\Gamma \in S_0$ denotes that for every variable v we have $\Gamma(v) \in S_0(v)$.

Proposition 1. *Let S be the set-based solution. Let v be any variable in P and let*

q be any qualifier in F. If $q \notin S(v)$ then there does not exist a valid typing $\Gamma \in S_0$, such that $\Gamma(v) = q$.

Proof. (Sketch) We say that q is a valid qualifier for v if there exists a valid typing Γ, where $\Gamma(v) = q$. Let v be the *first* variable that has a valid qualifier q removed from its set $S(v)$ and let $\textsc{SolveConstraint}(c)$ be the function that performs the removal. Since q is a valid qualifier there exist valid qualifiers $q_1, ..., q_k$ that make c satisfied. If $q_1 \in S(v_1)$ and $q_2 \in S(v_2)$, $\ldots$, and $q_k \in S(v_k)$, then by definition, $\textsc{SolveConstraint}(c)$ would not have had q removed from $S(v)$. Thus, one of $v_1, \ldots, v_k$ must have had a valid qualifier removed from its set *before* the application of $\textsc{SolveConstraint}(c)$. This contradicts the assumption that v is the first variable that has a valid qualifier removed.

The second proposition states that if we map every variable v to the maximal qualifier in its set $S(v)$ according to its preference ranking over qualifiers, and the typing is valid, then this typing maximizes the objective function.

Proposition 2. *Let o be the objective function over valid typings, and S be the set-based solution. A typing Γ is extracted as follows: $\Gamma(v) = max(S(v))$ for every variable v in P. If Γ is a valid typing, then Γ is a maximal typing of P in F under o.*

Proof. (Sketch) We show that Γ is a maximal typing. Suppose that there exists a valid typing $\Gamma' > \Gamma$. Let p_i be the most-preferred partition such that $\Gamma'^{-1}(p_i) \neq \Gamma^{-1}(p_i)$. Since $\Gamma' > \Gamma$, there must exist a variable v such that $\Gamma'(v) = q' \in p_i$, but $\Gamma(v) = q \notin p_i$. In other words, Γ' types v with $\Gamma'(v) = q' \in p_i$, but Γ types v differently — and lesser in the preference ranking, because $\Gamma'^{-1}(p_k) = \Gamma^{-1}(p_k)$ for $0 \leq k < i$ (here p_k are the more-preferred partitions than p_i). Since $\Gamma(v) = max(S(v))$, it follows that $q' \notin S(v)$. By Proposition 1, if $q' \notin S(v)$ there does not exist a valid typing which maps v to q', which contradicts the assumption that Γ' is a valid typing.

When each partition in the preference ranking has only a single element, then the weaker assumption "there exists a valid typing $\Gamma' \geq \Gamma$" can be contradicted, showing that the maximal typing is unique.

The *optimality property* holds for a type system F and a program P if and only if the typing derived from the set-based solution S by typing each variable with the maximally/preferred qualifier from its set, is a valid typing.

Property 1. *Optimality Property. Let F be a type system augmented with objective function o and let P be a program. The optimality property holds for F and P iff $\Gamma(v) = max(S(v))$, for all variables v, is a valid typing.*

Recall that the fixpoint can be reached in $O(n^2)$. Therefore, for type systems for which the optimality property holds for arbitrarily annotated programs, a maximal typing can be computed in quadratic time, with no manual annotations. If the programmer provides inconsistent initial annotations, the computation would terminate within $O(n^2)$ time with a list of type errors, meaning that a valid typing does not exist.

Remarkably, for several interesting systems (Reference Immutability Types, Universe Types, and more, see Chapter 3 and Chapter 5), the optimality property holds for unannotated programs, which means that the unique maximal typing can be computed in $O(n^2)$ time with no manual annotations. However, for Ownership Types and Information Flow Types, the optimality property does not hold. For these type systems we developed techniques that extract the best typing (see Chapter 4 and Chapter 6).

2.4 Type Checking

Type-checking verifies the correctness of the inferred typing. The type checker obtains the inferred typing Γ, and checks whether the typing fulfill the corresponding typing rule. This can be easily done using the Checker Framework [18].

For example, in ReIm, suppose the inferred typing Γ for statement $\mathsf{x} = \mathsf{y}$ is

$$\Gamma(\mathsf{x}) = \mathsf{mutable} \quad \Gamma(\mathsf{y}) = \mathsf{readonly}$$

According to (TASSIGN), it must satisfy $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x})$. However, this is not true because readonly is not a subtype of mutable. In this case, the type checker reports a type error and continues to the next statement.

In summary, we have presented the unified inference and checking framework for context-sensitivity pluggable types. In the following chapters, we instantiate the framework with several useful type systems, namely, ReIm, Universe Types, Ownership Types, SFlow/Integrity, and SFlow/Confidentiality.

CHAPTER 3
Reference Immutability Types

We have briefly discussed ReIm [15], a context-sensitive type system for reference immutability, as a running example in Chapter 2. In this chapter, we discuss the instantiation in detail.

An immutable, or readonly, reference cannot modify the state of an object, including the transitively reachable state. For instance, in the following code, the Date object cannot be modified by using the immutable reference rd, but the same Date object can be modified through the mutable reference md:

```
Date md = new Date();      // mutable by default
readonly Date rd = md;     // an immutable reference
md.setHours(1);            // OK, md is mutable
rd.setHours(1);            // compile-time error, rd is readonly
```

The type qualifier readonly denotes that rd is an immutable reference. By contrast to reference immutability, *object immutability* enforces a stronger guarantee that no reference in the system can modify a particular object. Each variety of immutability is preferable in certain situations [22]. This chapter only deals with reference immutability.

As a motivating example, consider a simplification of the Class.getSigners method which returns elements that have signed a particular class. In JDK 1.1, it is implemented approximately as follows:

```
class Class {
  private Object[] signers;
  public Object[] getSigners() {
    return signers;
  }
}
```

This implementation is not safe because a malicious client can obtain a reference to

Portions of this chapter previously appeared as: W. Huang *et al.*, “ReIm & ReImInfer: Checking and inference of reference immutability and method purity,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Tucson, AZ, 2012, pp. 879–896.

the signers array by invoking the getSigners method and can then side-effect the array to add an arbitrary trusted signer. Even though the field is declared private, the referenced object is still modifiable from the outside. There is no language support for preventing outside modifications, and the programmer must manually ensure that the code only returns clones of internal data.

A solution is to use reference immutability and annotate the return value of getSigners as readonly. (A readonly array of mutable objects is expressed, following Java 8 syntax [2], as Object readonly [].) As a result, mutations of the array through the returned reference will be disallowed:

```
Object readonly [] getSigners() {
  return signers;
}
...
Object readonly [] signers = getSigners();
signers[0] = maliciousClass; // compile-time error
```

A type system enforcing reference immutability has a number of benefits. It improves the expressiveness of interface design by specifying the mutability of parameters and return values; it helps prevent and detect errors caused by unwanted object mutations; and it facilitates reasoning about and proving other properties such as object immutability and method purity.

There are several type systems for Reference Immutability such as the capability system by Boyland et al. [5] and Javari by Tschantz and Ernst [6]. In this chapter, we instantiate the inference framework with our novel type system for reference immutability, ReIm, by giving the five framework parameters and defining the objective function (see Chapter 2).

3.1 Type Qualifiers and Subtying Relation

There are three source-level type qualifiers in ReIm: mutable, readonly, and polyread. These qualifiers were introduced by Javari [6] (except that polyread was romaybe in Javari but became polyread in Javarifier [16]). They have essentially the same meaning in Javari and ReIm, except that readonly in Javari allows certain fields and generic type arguments to be excluded from the immutability guarantee, while readonly in ReIm guarantees immutability of the entire structure.

- mutable: A mutable reference can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - x.f = z
  - x.setField(z) where setField sets a field of its receiver
  - y = id(x); y.f = z where id is a function that returns its argument
  - x.f.g = z
  - y = x.f; y.g = z
- polyread: A polyread reference x cannot be used to mutate the referenced object. However, x can be used to return the object, or anything it references, from m, and the object, or what it references, may be mutated after m returns. For example,
  - x.f = 0, where x is polyread, is not allowed, but
  - z = id(y); z.f = 0, where id is polyread X id(polyread X x) { return x; }, and y and z are mutable, is allowed.

  polyread expresses polymorphism (i.e., context sensitivity) over immutability. x's type may be interpreted as mutable in some call contexts, and it may also be interpreted as readonly in other call contexts.

  A polyread field is interpreted in the context of the receiver. For example, x.f, where f is polyread assumes the type of x — it is mutable when x is mutable and readonly when x is readonly. ReIm disallows mutable fields, i.e., fields can be only readonly or polyread. This is because the mutability of a reference reflects the mutability of its fields and allowing mutable fields would allow a reference to be typed readonly when obviously, some of its transitive state is mutated. This decision is motivated by our application of inferring pure methods (see

Section 3.7). Allowing mutable fields complicates the analysis of method purity. In addition, this is necessary to have the maximal typing type-checked for ReIm.

The subtyping relation between the qualifiers is

$$\mathsf{mutable} <: \mathsf{polyread} <: \mathsf{readonly}$$

For example, it is allowed to assign a mutable reference to a polyread or readonly one, but it is not allowed to assign a readonly reference to a polyread or mutable one.

3.2 Viewpoint Adaptation

Viewpoint adaptation interprets polyread in different contexts and expresses context sensitivity in ReIm.

3.2.1 Context Sensitivity

Consider the following code.

```
class DateCell {
  Date date;
  Date getDate(DateCell this) { return this.date; }
  void cellSetHours(DateCell this) {
    Date md = this.getDate();
    md.setHours(1);          // md is mutated
  }
  int cellGetHours(DateCell this) {
    Date rd = this.getDate();
    int hour = rd.getHours();  // rd is readonly
    return hour;
  }
}
```

In the above code, this of cellGetHours may be annotated as readonly, which is the top of the type hierarchy. Doing so is advantageous because then cellGetHours can be called on any argument.

The return value of method DateCell.getDate is used in a mutable context in cellSetHours and is used in a readonly context in cellGetHours. A context-insensitive type system would give the return type of getDate one specific type, which would have to be mutable. This would cause rd to be mutable, and then this of cellGetHours would have to be mutable as well (if this.date is of type mutable, this means that the current object was modified using this, which forces this to become mutable). This violates our goal that this of cellGetHours is readonly.

A context-sensitive type is required for the return type of DateCell.getDate. The effective return type will depend on the calling context. An example of calling context is the type of the left-hand side of a call assignment.

The polymorphic qualifier polyread expresses context sensitivity. We annotate this, the return type of getDate, and field date as polyread:

```
polyread Date date;
polyread Date getDate(polyread DateCell this) {
  return this.date;
}
```

Intuitively, viewpoint adaptation instantiates polyread to mutable in the context of cellSetHours, and to readonly in the context of cellGetHours. The call this.getDate on line 5 returns a mutable Date, and the call this.getDate on line 9 returns a readonly Date. As a result, the mutability of md propagates only to this of cellSetHours; it does not propagate to this of cellGetHours which remains readonly. ReIm handles polyread via viewpoint adaptation, and Javari/Javarifier handle polyread via templatizing methods. The two approaches appear to be semantically equivalent. Viewpoint adaptation however, is a more compact and scalable way of handling polymorphism than templatizing.

Conceptually, a method must type-check with each instance of polyread replaced by (adapted to) mutable, and with each instance of polyread replaced by readonly. Thus, a polyread reference x cannot be used to mutate the referenced object. A method may return x to the caller, in which case the caller might be able to mutate the object. Programmers should use polyread when the reference is readonly in the scope of the enclosing method, but may be modified in some caller contexts after the method's return.

The type of a polyread field f is adapted to the viewpoint of the receiver that accesses the field. If the receiver x is mutable, then x.f is mutable. If the receiver x is readonly, then x.f is readonly. If the receiver x is polyread, then x.f is polyread and cannot be used to modify the referenced object, as the access might be further instantiated with a readonly receiver. For example,

- x.f = 0, where x is polyread, is not allowed, but
- z = id(y); z.f = 0, where id is

  polyread X id(polyread X x) { return x; } ,

  is allowed when y and z are mutable.

We forbid mutable as a qualifier for fields. ReIm gives a strong reference immutability guarantee, including the whole transitive state. A mutable field would not depend on the type of the receiver and would therefore violate this guarantee.

3.2.2 Viewpoint Adaptation Operation

The viewpoint adaptation operation $\triangleright$ in ReIm is defined as:

$$\begin{aligned} \_ \triangleright \mathsf{mutable} &= \mathsf{mutable} \\ \_ \triangleright \mathsf{readonly} &= \mathsf{readonly} \\ q \triangleright \mathsf{polyread} &= q \end{aligned}$$

The underscore denotes a “don’t care” value. The operation means that mutable and readonly are independent of the context, while polyread is dependent on the context and is replaced by that context.

3.2.3 Context of Adaptation

For a field access, viewpoint adaptation $q \triangleright q_{\mathsf{f}}$ adapts the declared field qualifier q_{f} from the point of view of receiver qualifier q. In field access y.g where the field g is polyread, y.g takes the type of y. If y is readonly, then y.g must be readonly as well, in order to disallow modifications of y’s object through y.g. If y is polyread then y.g is polyread as well, propagating the context-dependency. Therefore, the function $\mathcal{C}$ for field access is defined as follows:

$$\begin{aligned} \mathcal{C}(\mathsf{y.f} = \mathsf{x}) &= q_{\mathsf{y}} \\ \mathcal{C}(\mathsf{x} = \mathsf{y.f}) &= q_{\mathsf{y}} \end{aligned}$$

For a method call $\mathsf{x} = \mathsf{y.m(z)}$, viewpoint adaptation $q_\mathsf{x} \rhd q$ adapts q, the declared qualifier of a formal parameter or the return value of m, from the point of view of q_x, the qualifier at the left-hand-side x of the call assignment. If a formal parameter or the return value is $\mathsf{readonly}$ or $\mathsf{mutable}$, its adapted type remains the same regardless of q_x. However, if q is $\mathsf{polyread}$, the adapted type depends on q_x — it becomes q_x (i.e., the $\mathsf{polyread}$ type is the polymorphic type, and it is instantiated to q_x). Thus, the context of adaptation at a method call is the *left-hand-side of the call assignment* x and the function $\mathcal{C}$ is defined for method call as:

$$\mathcal{C}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) \quad = \quad q_\mathsf{x}$$

In the case that there is no return value, i.e. the return value is void, ReIm assumes that the context of adaptation is $\mathsf{readonly}$.

3.3 Additional Constraints

ReIm only imposes additional constraints for (TWRITE), which requires that the receiver of the field write must be $\mathsf{mutable}$. Thus, we define the additional constraints function $\mathcal{B}$ as:

$$\mathcal{B}(\mathsf{y.f} = \mathsf{x}) \quad = \quad \{\mathsf{y} = \mathsf{mutable}\}$$

$\mathcal{B}$ returns empty set for all other statements.

3.4 Instantiated Typing Rules

By giving the above five framework parameters, the unified typing rules in Figure 2.3 are instantiated for ReIm. The instantiated typing rules are shown in Figure 3.1.

These typing rules impose subtying constraints, viewpoint adaptation, and additional constraints as discussed in Chapter 2. Rule (TCALL) demands a detailed explanation. It requires $q_\mathsf{x} \rhd q_\mathsf{ret} <: q_\mathsf{x}$. This constraint disallows the return value of m from being $\mathsf{readonly}$ when there is a call to m, $\mathsf{x} = \mathsf{y.m(z)}$, where left-hand-side x is $\mathsf{mutable}$. Only if the left-hand-sides of all call assignments to m are $\mathsf{readonly}$, can the return type of m be $\mathsf{readonly}$; otherwise, it is $\mathsf{polyread}$. A programmer can

$$\overset{\text{(TNEW)}}{\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}} \qquad \overset{\text{(TASSIGN)}}{\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}}$$

$$\overset{\text{(TWRITE)}}{\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ q_{\mathsf{x}}\ <:\ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \\ \mathcal{B}(\mathsf{y.f} = \mathsf{x}) = \{q_{\mathsf{y}} = \mathsf{mutable}\}\end{array}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}} \qquad \overset{\text{(TREAD)}}{\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{y}} \rhd q_{\mathsf{f}}\ <:\ q_{\mathsf{x}} \\ \end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}}$$

$$\overset{\text{(TCALL)}}{\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}}\ <:\ q_{\mathsf{x}} \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}}\ <:\ q_{\mathsf{x}} \rhd q_p \quad q_{\mathsf{x}} \rhd q_{\mathsf{ret}}\ <:\ q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y}.\mathsf{m}^i(\mathsf{z})}}$$

Figure 3.1: Instantiated Typing Rules for ReIm.

annotate the return type of m as mutable. However, this typing is pointless, because it unnecessarily forces local variables and parameters in m to become mutable when they can be polyread.

In addition, the rule requires $q_{\mathsf{y}}\ <:\ q_{\mathsf{x}} \rhd q_{\mathsf{this}}$. When q_{this} is readonly or mutable, its adapted value is the same. Thus, when q_{this} is mutable (e.g., due to this.f = 0 in m),

$$q_{\mathsf{y}}\ <:\ q_{\mathsf{x}} \rhd q_{\mathsf{this}} \quad \text{becomes} \quad q_{\mathsf{y}}\ <:\ \mathsf{mutable}$$

which disallows q_{y} from being anything but mutable, as expected. The most interesting case arises when q_{this} is polyread. Recall that a polyread parameter this is readonly within the enclosing method, but there could be a dependence between this and ret such as

$$\mathsf{X\ m()}\ \{\ \mathsf{z = this.f; w = z.g; return\ w;}\ \}$$

which allows the this object to be modified in caller context, after m's return. Well-formedness guarantees that whenever there is dependence between this and ret, as in the above example, the following constraint holds:

$$q_{\mathsf{this}} <: q_{\mathsf{ret}}$$

Recall that when there exists a context where the left-hand-side variable x is mutated, q_{ret} must be $\mathsf{polyread}$. Therefore, constraint $q_{\mathsf{this}} <: q_{\mathsf{ret}}$ forces q_{this} to be $\mathsf{polyread}$ (let us assume that this is not mutated in the context of its enclosing method).

The role of viewpoint adaptation is to transfer the dependence between this and ret in m, into a dependence between actual receiver y and left-hand-side x in the call assignment. In the above example, there is a dependence between this and the return ret. Thus, we also have a dependence between y and x in the call $\mathsf{x} = \mathsf{y.m}()$ — that is, a mutation of x makes y mutable as well. Function $\rhd$ does exactly that. Rule (TCALL) requires

$$q_{\mathsf{y}} \;<:\; q_{\mathsf{x}} \rhd q_{\mathsf{this}}$$

when there is a dependence between this and ret, q_{this} is $\mathsf{polyread}$, and the above constraint becomes

$$q_{\mathsf{y}} \;<:\; q_{\mathsf{x}}$$

This is exactly the constraint we need. If x is mutated, y becomes $\mathsf{mutable}$ as well. In contrast, if x is $\mathsf{readonly}$, y remains unconstrained.

3.5 Type Inference

We leverage the inference framework for type inference of ReIm. Because the maximal typing for ReIm provably type-checks (see proof below), we only need to define (1) the initial set mapping S_0, and (2) the objective function o_{ReIm}.

3.5.1 Initial Mapping

The set mapping S_0 is initialized as follows. Programmer-annotated variables, if any, are initialized to the singleton set that contains the programmer-provided type. Note that there might be no programmer-annotated variables, because the inference of ReIm works without any input from programmers. Method returns are initialized $S(\mathsf{ret}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ for each method m. Fields are initialized $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{polyread}\}$. All other variables are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$.

3.5.2 Objective Function

As discussed in Chapter 2, the objective function for ReIm is defined as

$$o_{ReIm}(\Gamma) = (|\Gamma^{-1}(\mathsf{readonly})|, |\Gamma^{-1}(\mathsf{polyread})|, |\Gamma^{-1}(\mathsf{mutable})|)$$

The partitioning and ordering is

$$\{\mathsf{readonly}\} > \{\mathsf{polyread}\} > \{\mathsf{mutable}\}$$

This means, informally, that we prefer readonly over polyread and mutable, and polyread over mutable. More formally, the partitioning and ordering gives rise to a preference ranking O_{ReIm} over all qualifiers:

$$O_{ReIm} : \mathsf{readonly} > \mathsf{polyread} > \mathsf{mutable}$$

This ranking maximizes the number of readonly references. Note that leaving all references as mutable is also a valid typing but a useless one, as it expresses nothing about immutability.

3.5.3 Maximal Typing

By picking the maximal qualifier from each set of variables, the resulting maximal typing for ReIm is correct, precise, and maximal. The following propositions formalize its properties.

Proposition 3. *The maximal typing type-checks under the rules from Figure 3.1.*

Proof. (Sketch) The proof is a case-by-case analysis which shows that after the application of the $\textsc{SolveConstraint}(c)$ function for each constraint c in statement s, the rule type-checks with the maximal assignment on statement s. Let $max(S(\mathsf{x}))$ return the maximal element of $S(\mathsf{x})$ according to the preference ranking (which is the same as the type hierarchy). We show (TCALL) $\mathsf{x} = \mathsf{y.m(z)}$. The rest of the cases are straightforward.

- Let $max(S(\mathsf{x}))$ be $\mathsf{readonly}$.

  If $max(S(\mathsf{this}))$ is $\mathsf{readonly}$ or $\mathsf{polyread}$, $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(\mathsf{this}))$ is $\mathsf{mutable}$, the only possible max for y would be $\mathsf{mutable}$ (the others would have been removed by SOLVECONSTRAINT(c) where c is $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$). $q_\mathsf{z} \ <:\ q_\mathsf{x} \rhd q_p$ is analogous to $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$. $q_\mathsf{x} \rhd q_\mathsf{ret} \ <:\ q_\mathsf{x}$ holds for any value of $max(S(\mathsf{ret}))$.

- Let $max(S(\mathsf{x}))$ be $\mathsf{mutable}$.

  If $max(S(\mathsf{this}))$ is $\mathsf{readonly}$, $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(\mathsf{this}))$ is $\mathsf{polyread}$, the only possible value for $max(S(\mathsf{y}))$ would be $\mathsf{mutable}$. If $max(S(\mathsf{this}))$ is $\mathsf{mutable}$, the only possible max for y would be $\mathsf{mutable}$ as well (the others would have been removed by SOLVECONSTRAINT(c) where c is $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$).

  If $max(S(\mathsf{ret}))$ is $\mathsf{polyread}$, clearly $q_\mathsf{x} \rhd q_\mathsf{ret} \ <:\ q_\mathsf{x}$ holds. $max(S(\mathsf{ret}))$ cannot be $\mathsf{readonly}$, $\mathsf{readonly}$ would have been removed by the function.

- Let $max(S(\mathsf{x}))$ be $\mathsf{polyread}$.

  If $max(S(\mathsf{this}))$ is $\mathsf{readonly}$, $q_\mathsf{y} \ <:\ q_\mathsf{x} \rhd q_\mathsf{this}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(q_\mathsf{this}))$ is $\mathsf{polyread}$, the only possible values for $max(S(\mathsf{y}))$ would be $\mathsf{polyread}$ or $\mathsf{mutable}$. If $max(S(q_\mathsf{this}))$ is $\mathsf{mutable}$, the only possible max for y would be $\mathsf{mutable}$.

  If $max(S(\mathsf{ret}))$ is $\mathsf{polyread}$, clearly $q_\mathsf{x} \rhd q_\mathsf{ret} \ <:\ q_\mathsf{x}$ holds. $max(S(\mathsf{ret}))$ cannot be $\mathsf{readonly}$, $\mathsf{readonly}$ would have been removed by the function.

Recall Proposition 2 (see Chapter 2) which states that a valid maximal typing maximizes the objective function. Since the maximal typing for ReIm is valid according to Proposition 3, it maximizes the objective function o_{ReIm}. In addition, the maximal typing is unique. Thus, the optimality property holds and we have found the best typing.

```
class A {
  X f;                      S(f) = {polyread}
  X get(A this, Y y) {      S(this_get) = {polyread, mutable}
    ... = y.h;              S(y_get) = {readonly, polyread, mutable}
    X x = this.getF();      S(x_get) = {polyread, mutable}
    return x;               S(ret_get) = {polyread}
  }
  X getF(A this) {          S(this_getF) = {polyread, mutable}
    X x = this.f;           S(x_getF) = {polyread, mutable}
    return x;               S(ret_getF) = {polyread}
  }
}
void setG() {
  A a = ...                 S(a_setG) = {mutable}
  Y y = ...                 S(y_setG) = {readonly, polyread, mutable}
  X x = a.get(y);           S(x_setG) = {mutable}
  x.g = null;
}
void getG() {
  A a = ...                 S(a_getG) = {readonly, polyread, mutable}
  Y y = ...                 S(y_getG) = {readonly, polyread, mutable}
  X x = a.get(y);           S(x_getG) = {readonly, polyread, mutable}
  ... = x.g;
}
```

Figure 3.2: Inference Example for ReIm. A.get(Y) has different mutabilities in the contexts of setG and getG. Also, A.getF(), which is called from A.get(Y), has different mutabilities in different calling contexts. The box beside each statement shows the set-based solution; the underlined qualifiers are the final qualifiers picked by ReIm.

3.6 Inference Example

Consider the example in Figure 3.2. We use $\mathsf{x}_{\mathsf{get}}$ to denote the reference x in method get. Initially, all references are initialized to the sets as described above. The analysis iterates over all statements in class A and in methods setG and getG. In the first iteration, the analysis changes nothing until it processes x.g = null in setG. $S(\mathsf{x}_{\mathsf{setG}})$ is updated to $\{\mathsf{mutable}\}$. In the second iteration, when the analysis processes x = a.get(y) in setG, $S(\mathsf{ret}_{\mathsf{get}})$ becomes $\{\mathsf{polyread}\}$. In the third iteration, $S(\mathsf{x}_{\mathsf{get}})$ becomes $\{\mathsf{polyread}, \mathsf{mutable}\}$ because $\mathsf{x}_{\mathsf{get}}$ has to be a subtype of $S(\mathsf{ret}_{\mathsf{get}})$. This in turn forces $S(\mathsf{ret}_{\mathsf{getF}})$ and subsequently $S(\mathsf{this}_{\mathsf{getF}})$ to become $\{\mathsf{polyread}, \mathsf{mutable}\}$. The iteration continues until it reaches the fixpoint as shown in the boxes in Figure 3.2. For brevity, some references are not shown in the boxes. The underlined qualifiers are the maximal element in the preference ranking.

3.7 Method Purity Inference

In this section, we present method purity inference built as extension of ReIm.

A method is *pure* (or side-effect free) when it has no visible side effects. Knowing which methods are pure has a number of practical applications. It can facilitate compiler optimization [23, 24, 25], model checking [26], Universe Type inference [17, 27], memoization of function calls [28], and so on.

We adopt the definition of purity given by Sălcianu and Rinard [29]: a method is *pure* if it does not mutate any object that exists in *prestates*. Thus, a method is pure if (1) it does not mutate prestates reachable through parameters, and (2) it does not mutate prestates reachable through static fields. The definition allows a pure method to create and mutate local objects, as well as to return a newly constructed object as a result. This is the semantics of the @Pure annotation in JML.

For a method that does not access static fields, the prestates it can reach are the objects reachable from the actual arguments and the method receiver. Therefore, if any of the formal parameters of m or implicit parameter this is inferred as mutable by reference immutability inference, m is impure. Otherwise, i.e., if none of the parameters is inferred as mutable, m is pure. Consider the implementation of List in the left column of Figure 3.3. For method add, reference immutability inference infers that both n and this are mutable, i.e., the objects referred by them may be mutated in add. When there is a method invocation lst.add(node), we know that the prestates referred to by the actual argument node and the receiver lst may be mutated. As a result, we can infer that method add is impure. We can also infer

```
class List {
  Node head;
  int len;
  void add(Node n) {
    n.next = this.head;
    this.head = n;
    this.len++;
  }
  void reset() {
    this.head = null;
    this.len = 0;
  }
  int size() {
    return this.len;
  }
}
```

```
class Main {
  static List sLst;
  void m1() {
    List lst = ...
    Node node = ...
    lst.add(node);
    Main.sLst = lst;
  }
  void m2() {
    int len = sLst.size();
    PrintStream o = System.out;
    o.print(len);
  }
  void m3() {
    m2();
  }
}
```

Figure 3.3: A simple linked list and example usage.

that method reset is impure because implicit parameter this is inferred as mutable by reference immutability inference. Method size is inferred as pure because its implicit parameter this is inferred as readonly and it has no formal parameters.

However, the prestates can also come from static fields. A method is impure if it mutates (directly, or indirectly through callees), a static field, or objects reachable from a static field. We introduce a *static immutability type* q_{m} for each method m. Roughly, q_{m} is mutable when m accesses static state through some static field and then mutates this static state; q_{m} is polyread if m accesses static state but does not mutate this state directly, however, m may return this static state to the caller and the caller may mutate it; q_{m} is readonly otherwise. Static immutability types are computed using reference immutability. We introduce a function *statictypeof* that retrieves the static immutability type of m:

$$\textit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}}$$

We extend the program syntax with two additional statements (TSWRITE) sf = x for static field write, and (TSREAD) x = sf for static field read. Here x denotes a local

$$\frac{\begin{array}{c} \textit{methodof}(\mathsf{sf} = \mathsf{x}) = \mathsf{m} \quad \textit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}} \quad q_{\mathsf{m}} = \mathsf{mutable} \\ \Gamma(\mathsf{sf}) = q_{\mathsf{sf}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{sf}} \end{array}}{\Gamma \vdash \mathsf{sf} = \mathsf{x}} \ \textsc{(tswrite)}$$

$$\frac{\begin{array}{c} \textit{methodof}(\mathsf{x} = \mathsf{sf}) = \mathsf{m} \quad \textit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}} \\ \Gamma(\mathsf{sf}) = q_{\mathsf{sf}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{m}} <: q_{\mathsf{x}} \quad q_{\mathsf{sf}} <: q_{\mathsf{x}} \end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{sf}} \ \textsc{(tsread)}$$

$$\frac{\begin{array}{c} \Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \quad \textit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \\ q_{\mathsf{y}} \ <: \ q_{\mathsf{x}} \rhd q_{\mathsf{this}} \qquad q_{\mathsf{z}} \ <: \ q_{\mathsf{x}} \rhd q_p \quad q_{\mathsf{x}} \rhd q_{\mathsf{ret}} \ <: \ q_{\mathsf{x}} \\ \underline{\textit{methodof}(\mathsf{x} = \mathsf{y.m(z)}) = \mathsf{m}'} \quad \underline{\textit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}}} \\ \underline{\textit{statictypeof}(\mathsf{m}') = q_{\mathsf{m}'}} \quad \underline{q_{\mathsf{m}'} \ <: \ q_{\mathsf{x}} \rhd q_{\mathsf{m}}} \end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}} \ \textsc{(tcall)}$$

Figure 3.4: Extended typing rules for static fields (see Figure 3.1 for the base type system). Function *methodof*(s) returns the enclosing method of statement s. Function *statictypeof*(m) returns the static immutability type of m. Static immutability types can be readonly, polyread, or mutable. Rule (TCALL) includes the antecedents from the base type system and the new antecedents that handle the static immutability type of method m, underlined.

variable and sf denotes a static field.

In contrast to instance fields, static fields are declared as either readonly or mutable. There is no receiver for static field accesses and therefore no substitution for polyread would occur.

Figure 3.4 extends the typing rules from Figure 3.1 with constraints on static immutability types. If method m contains a static field write sf = x, then its static immutability type is mutable (see rule (TSWRITE)). If m contains a static field read x = sf where x is inferred as mutable, q_{m} becomes mutable as well (see rule (TSREAD)). While the handling of (TSWRITE) is expected, the handling of (TSREAD) may be unexpected. If sf is read in m, using x = sf, then m or one of its callees can access and mutate the fields of sf through x. If m or one of its callees writes a field of sf through x, then x will be mutable. If m does not write x, but returns x to a caller and the caller subsequently writes a field of the returned object, then x will be polyread. x being readonly guarantees that x is immutable in the scope of m and after m's return, and sf is not mutated through x. Note that aliasing is handled by the type system which disallows assignment from readonly to mutable or polyread. Consider the code:

```
void m() {
  ...
  x = sf; // a static field read
  y = x.f;
  z = id(y);
  z.g = 0;
  ...
}
```

Here static field sf has its field f aliased to local z, which is mutated. The type system propagates the mutation of z to x; thus, the constraints in Figure 3.4 set the static immutability type of m to mutable.

Rule (TCALL) in Figure 3.4 captures two cases:

1. If the callee m mutates static fields, i.e. *statictypeof*(m) = mutable, the *statictypeof*(m′) of enclosing method m′ has to be mutable as well, because $\_ \rhd \mathsf{mutable} = \mathsf{mutable}$.

2. If the callee m returns a static field which is mutated later, *statictypeof*(m) would be polyread. If the enclosing method m′ mutated the return value x, i.e. $q_{\mathsf{x}} = \mathsf{mutable}$, *statictypeof*(m′) would be mutable because $\mathsf{mutable} \rhd \mathsf{polyread} = \mathsf{mutable}$. Otherwise, if x is polyread, *statictypeof*(m′) is constrained to polyread or mutable. Finally, if x is readonly, *statictypeof*(m′) is readonly, indicating that m′ does not mutate static fields.

Method overriding is handled by an additional constraint. If m′ overrides m we must have

$$q_{\mathsf{m}} <: q_{\mathsf{m'}}$$

In other words, if m′ mutates static state, q_{m} must be mutable, even if m itself does not mutate static state. This constraint ensures that m′ is a behavioral subtype of m and is essential for modularity.

Static immutability types are inferred in the same fashion as reference immutability types. The analysis initializes every $S(\mathsf{m})$ to $\{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$

and iterates over the constraints generated by the statements in Figure 3.4 and the overriding constraints, until it reaches the fixpoint. If readonly remains in $S(\mathsf{m})$ at the end, the static immutability type of m is readonly; otherwise, it is polyread or mutable.

Consider the right column of Figure 3.3. q_{m1} becomes mutable because m1 assigns lst to the static field sLst. q_{m2} is mutable as well, because it mutates the PrintStream object referred by System.out by invoking the print method on it (note that the implicit this parameter of print is mutable), and local variable o is mutable. q_{m3} becomes mutable as well, because it invokes method m2 and q_{m2} is mutable. Since q_{m1}, q_{m2}, and q_{m3} are mutable, this implies that these methods have mutated the presents through static fields.

The observant reader has likely noticed that $q_{\mathsf{m}} =$ mutable does not account for all mutations of static state in m. In particular, static state may be aliased to parameters and be accessed and mutated in m through parameters:

```
void m(X p) {
  p.g = 0;
}
...
void n() {
  X x = sf; // a static field read (TSREAD)
  m(x);
}
```

In the above example, q_{m} is readonly, even though m mutates static state. Interestingly, this is not unsound. Note however that method m is impure, because there is a write to its parameter. Parameter and static mutability types capture precisely the information needed to infer purity as we shall see shortly.

We infer that a method m is pure if all of its parameters, including implicit parameter this, are not mutable (i.e., they are readonly or polyread), and its static immutability type is not mutable (i.e., it is readonly or polyread). More formally, let

$\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}}$ and $\mathit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}}$. We have:

$$\mathit{pure}(\mathsf{m}) = \begin{cases} \textbf{false} & \textbf{if } q_{\mathsf{this}} = \mathsf{mutable}\ \textbf{or} \\ & \quad q_p = \mathsf{mutable}\ \textbf{or} \\ & \quad q_{\mathsf{m}} = \mathsf{mutable} \\ \textbf{true} & \textbf{otherwise} \end{cases}$$

As discussed earlier, a method m can be impure because: (1) prestates are mutated through parameters, or (2) prestates are mutated through static fields. If prestates are mutated through parameters, then this will be captured by the mutability of this and p. Now, suppose that prestates are *not* mutated through parameters, but are mutated after access through a static field. In this case, there must be an access in m to a static field sf through (TSREAD) or (TSWRITE), and the mutation is captured by the static immutability type q_{m}.

CHAPTER 4
Ownership Types

Aliasing is a powerful feature of object-oriented programming as it makes sharing states among objects easier. However, it is also a weakness of object-oriented programming [7]. When an object's internal state is aliased, it loses full control of its own state; thus, it cannot prevent unwanted mutation from happening. This imposes the following problems for object-oriented programming:

- Bugs are difficult to detect. Because aliasing leads to cross references among different objects, it is very hard to find out which parts of an object are potentially changed by the code under consideration [30].
- Automatic memory management becomes difficult. The life time of an object is hard to track because of aliasing.
- Modular reasoning about programs becomes difficult. Without any control or knowledge of the aliasing and effect, modular reasoning is not possible [31, 32].

The *Geneva Convention on Treatment of Object Aliasing* categorizes approaches for treating object aliasing [33]:

- Detection. Static or dynamic (run-time) diagnosis of potential or actual aliasing.
- Advertisement. Annotations that help modularize detection of aliasing by declaring aliasing properties of methods.
- Prevention. Constructs that disallow aliasing in a statically checkable fashion.
- Control. Mechanisms that isolate the effects of aliasing.

Many researchers have explored different approaches for treating aliasing. Object ownership, as a prevention mechanism, has been successfully applied in

Portions of this chapter previously appeared as: W. Huang *et al.*, "Inference and checking of object ownership," in *Proc. European Conf. Object-Oriented Programming*, Beijing, China, 2012, pp. 181–206.

prevention and detection of data races [34, 35], efficient message passing [36], modular reasoning [31, 37], and more.

Recall the getSigners example in Chapter 3. The keyword private only provides name-based protection on the internal array signers. Thus, this private information can still be leaked out when clients invoke getSigners(). In contrast, ownership provides control-based protection. With object ownership, the getSigners bug can be detected in the compilation stage if the programmer annotated the internal array as rep, which denotes that the reference is owned by the current object and it cannot be exposed to the outside world.

```
class Class {
  private Object rep [] signers;
  public Object rep [] getSigners() {
    return signers;
  }
}
```

As a result, the following code in clients will fail type-checking at compile time:

```
Object rep[] a = c.getSigners();
```

because rep of a and rep of signers refer to different owners: a is owned by the current this object but signers is owned by the c object. A solution is to use ownership types and annotate the internal array signers as rep. As a result, the internal representation signers of the Class object cannot be exposed to the outside, thus protecting the signers from unwanted modifications from outside.

There are many different ownership type systems in the literature [7, 8, 38, 39, 40]. Most of them enforce the *owner-as-dominator* discipline: all accesses to an object have to go through its owner. In other words, the owner has full control of the (transitively) owned objects. Recall the above getSigners example. When the signers is annotated as rep, all accesses to the array have to go through its owner, which is the Class object. Return of the signers reference through invocation of c.getSigners() is not allowed.

In this chapter, we instantiate the inference framework with the classical Ownership Types (OT) [7] which enforce the *owner-as-dominator* discipline. We illustrate how OT can be specified, inferred and checked in our framework. We restrict OT to one ownership parameter, as our experience with real world programs suggests that one ownership parameter is sufficient in practice (see Section 7.2.3).

4.1 Type Qualifiers and Subtying Relation

There are three base ownership modifiers in Ownership Types:

- rep refers to the current object this.
- own refers to the owner of the current object.
- p is an ownership parameter passed to the current object.

OT qualifiers have the form $\langle q_0|q_1\rangle$, where q_0 and q_1 are one of rep, own, or p. A qualifier $\langle q_0|q_1\rangle$ for reference variable x is interpreted as follows. Let i be the object referenced by x. q_0 is the *owner* of i, from the point of view of the current object, and q_1 is the *ownership parameter* of i, again, from the point of view of the current object. Informally, the ownership parameter q_1 refers to an object, which objects referenced by i might use as owner. For example, ⟨rep|own⟩ x means that the owner of i is the current object this, and the ownership parameter passed to i is the owner of the current object. Transitively, objects referenced by i, for example, from its fields, can have as owner (1) i itself, by using rep, (2) the current object, by using own, or (3) the owner of the current object, by using p.

There are six type qualifiers in OT: ⟨rep|rep⟩, ⟨rep|own⟩, ⟨rep|p⟩, ⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩. These ownership types give rise to an encapsulation structure called *ownership tree*, where nodes denote objects and the parent of a node denotes its owner.

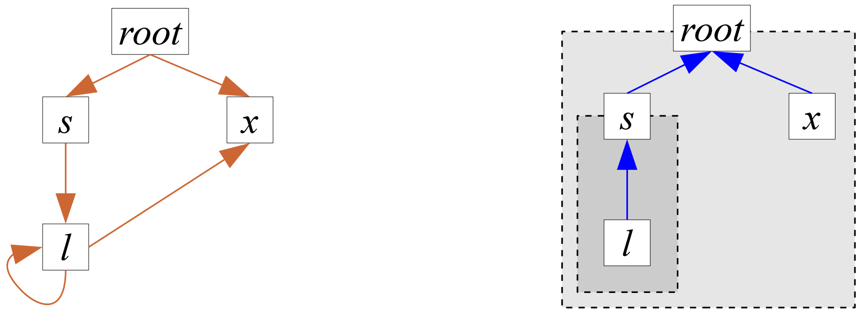
There is no subtyping relation between these type qualifiers, which means both sides of an assignment (explicit or implicit) must have the same type qualifier.

Figure 4.1 shows the XStack program from [7] annotated with Ownership Types. Link contains two fields: next and data. next (line 21) is of type ⟨own|p⟩ which means that all Link objects have the same owner. data (line 22) is of type ⟨p|p⟩ which can be bound to the ownership parameter of XStack. This means that the data is not necessarily owned by Link or XStack.

```
 1  class XStack {
 2    ⟨rep|p⟩ Link top;
 3    XStack() {
 4      top = null;
 5    }
 6    void push(⟨p|p⟩ X d1) {
 7      ⟨rep|p⟩ Link newTop;
 8      newTop = new ⟨rep|p⟩ Link(); $\boxed{l}$
 9      newTop.init(d1);
10      newTop.next = top;
11      top = newTop;
12    }
13    void main(String[] arg) {
14      ⟨rep|rep⟩ XStack s;
15      s = new ⟨rep|rep⟩ XStack(); $\boxed{s}$
16      ⟨rep|rep⟩ X x = new ⟨rep|rep⟩ X(); $\boxed{x}$
17      s.push(x);
18    }
19  }
20  class Link {
21    ⟨own|p⟩ Link next;
22    ⟨p|p⟩ X data;
23    void init(⟨p|p⟩ X d2) {
24      next = null;
25      data = d2;
26    }
27  }
```

Figure 4.1: Annotated XStack with Ownership Types. The boxed italic letters denote object allocation sites.

XStack is built from a singly-linked list of Link nodes. Because we do not know the owners of the data elements, the data (line 22) is typed as ⟨p|p⟩ which can be instantiated by the actual owners of the data elements (it is bound to the *root* in this example). This is similar to generic classes with parametric polymorphism — the XStack is reusable because of the ownership parameter. At line 8, a new instance of Link is typed as ⟨rep|p⟩, which means its owner is the current XStack object. Interestingly, all neighbours of the new Link instance are owned by the same XStack instance. Because newTop.next and newTop have the same owner (recall next is of



(a) Object graph

(b) Ownership tree of XStack in OT

Figure 4.2: Object graph (left) and ownership tree (right) for XStack for the example in Figure 4.1. In the object graph we show all references between objects. In the ownership tree we draw an arrow from the owned object to its owner and put all objects with the same owner into a dashed box.

type ⟨own|p⟩), and the owner of newTop is the current XStack object, we conclude that the current XStack object is also the owner of newTop.next. Using this same argument, we know the XStack object owns all the links, which means all accesses to the links have to go through their owner — the XStack object. An XStack instance s and an X instance x are constructed in the static main method with ⟨rep|rep⟩ annotation at line 15 and 16, respectively.

Figure 4.2 shows the object graph and the corresponding ownership tree. The s object and the x object are owned by the *root*. And the Link object l is owned by the s object.

4.2 Viewpoint Adaptation

Viewpoint adaptation in Ownership Types encodes the relation between the current this object and its transitively reachable objects.

4.2.1 Viewpoint Adaptation Operation

In Ownership Types, viewpoint adaptation is applied only when the adaptation context (q in $q \triangleright q'$) is *not* this. This is because the ownership context does not change at field access and method call when the receiver is this. Therefore, we introduce the type qualifier self. Any type q adapting from the context self yields q itself, i.e.

$\mathsf{self} \rhd q = q$. Note that self is used only internally, to distinguish field access and method calls through this from ones not through this. Programmers cannot annotate any reference as self. The default type for the implicit parameter this is $\langle \mathsf{own}|\mathsf{p}\rangle$.

Viewpoint adaptation operation of Ownership Types is defined as follows:

$$\begin{array}{rcl}
\langle q_0|q_1\rangle \rhd \langle \mathsf{own}|\mathsf{own}\rangle & = & \langle q_0|q_0\rangle \\
\langle q_0|q_1\rangle \rhd \langle \mathsf{own}|\mathsf{p}\rangle & = & \langle q_0|q_1\rangle \\
\langle q_0|q_1\rangle \rhd \langle \mathsf{p}|\mathsf{p}\rangle & = & \langle q_1|q_1\rangle \\
\mathsf{self} \ \rhd \langle q_0|q_1\rangle & = & \langle q_0|q_1\rangle
\end{array}$$

Viewpoint adaptation disallows that the adapted type contains rep, which accounts for the static visibility constraint [7].

As an example, let us discuss the first rule: the adapted type of $\langle \mathsf{own}|\mathsf{own}\rangle$ from the point of view of $\langle q_0|q_1\rangle$ is $\langle q_0|q_0\rangle$. If an object i has type $\langle q_0|q_1\rangle$ from the point of view of the current this object, this means that the owner of i is q_0. If object j has type $\langle \mathsf{own}|\mathsf{own}\rangle$ from the point of view of i, this means that both j's owner and ownership parameter are instantiated to the *owner* of i. Therefore, j will have type $\langle q_0|q_0\rangle$ from the point of view of this.

4.2.2 Context of Adaptation

Recall that no viewpoint adaptation is needed when the receiver is this at field access or method call. Therefore, the context of adaptation function $\mathcal{C}$ returns self when the receiver is this, and returns the type of the receiver otherwise:

$$\mathcal{C}(\mathsf{y.f} = \mathsf{x}) \quad = \quad \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}(\mathsf{x} = \mathsf{y.f}) \quad = \quad \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) \quad = \quad \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

For example, consider a field read $\mathsf{x} = \mathsf{y.f}$ where $\mathsf{y} \neq \mathsf{this}$. $\mathcal{C}(\mathsf{x} = \mathsf{y.f})$ returns q_{y}. Let y have type $q_{\mathsf{y}} = \langle \mathsf{rep}|\mathsf{rep} \rangle$ and let field f have type $q_{\mathsf{f}} = \langle \mathsf{own}|\mathsf{p} \rangle$. Then $\mathsf{y.f}$ has type $\langle \mathsf{rep}|\mathsf{rep} \rangle$. The first rep in this type can be explained as follows: Owner own in the type of f gives us that the owner of the f object is the same as the owner of the y object, and owner rep in the type of y gives us that the owner of the y object is the current this object. Thus, the owner of the f object, from the point of view of the current object, is the current object.

4.2.3 Additional Constraints

In Ownership Types, all $\mathcal{B}$ sets are empty as the system does not impose additional constraints beyond the standard subtyping and viewpoint adaptation constraints. Note that the subtyping constraints degenerate into equality constraints as OT does not have a subtyping relation.

4.3 Instantiated Typing Rules

The instantiated typing rules for Ownership Types are shown in Figure 4.3. Note that for readability, we separate the typing rules (TWRITE), (TREAD), and (TCALL) when the receiver is this. The instantiated typing rules still fit into the framework.

4.4 Type Inference

In this section, we first define the initial set mapping S_0 and the objective function o_{OT} to leverage the inference framework for type inference of Ownership Types. Then we discuss the maximal typing for Ownership Types.

4.4.1 Initial Mapping

S_0 is initialized as follows. Programmer-annotated variables are initialized to the singleton set of the provided type. The implicit parameter this is initialized $S(\mathsf{this}) = \{\langle \mathsf{own}|\mathsf{p} \rangle\}$. Library variables are initialized to $\{\langle \mathsf{own}|\mathsf{p} \rangle, \langle \mathsf{p}|\mathsf{p} \rangle\}$ in order to handle containers from Java library. All other variables are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{rep} \rangle, \langle \mathsf{rep}|\mathsf{own} \rangle, \langle \mathsf{rep}|\mathsf{p} \rangle, \langle \mathsf{own}|\mathsf{own} \rangle, \langle \mathsf{own}|\mathsf{p} \rangle, \langle \mathsf{p}|\mathsf{p} \rangle\}$.

(TNEW)

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}$$

(TASSIGN)

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

(TWRITE)

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ q_{\mathsf{x}} \ <: \ q_{\mathsf{y}} \triangleright q_{\mathsf{f}}\end{array}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

(TREAD)

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{y}} \triangleright q_{\mathsf{f}} \ <: \ q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

(TWRITETHIS)

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{x}} \ <: \ q_{\mathsf{f}}\end{array}}{\Gamma \vdash \mathsf{this.f} = \mathsf{x}}$$

(TREADTHIS)

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{f}} \ <: \ q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{this.f}}$$

(TCALL)

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} \ <: \ q_{\mathsf{y}} \triangleright q_{\mathsf{this}} \quad q_{\mathsf{z}} \ <: \ q_{\mathsf{y}} \triangleright q_p \quad q_{\mathsf{y}} \triangleright q_{\mathsf{ret}} \ <: \ q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}$$

(TCALLTHIS)

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} \ <: \ q_{\mathsf{this}} \quad q_{\mathsf{z}} \ <: \ q_p \quad q_{\mathsf{ret}} \ <: \ q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{this.m}^i(\mathsf{z})}$$

Figure 4.3: Instantiated Typing Rules for Ownership Types. For readability we include separate typing rules when the receiver is this.

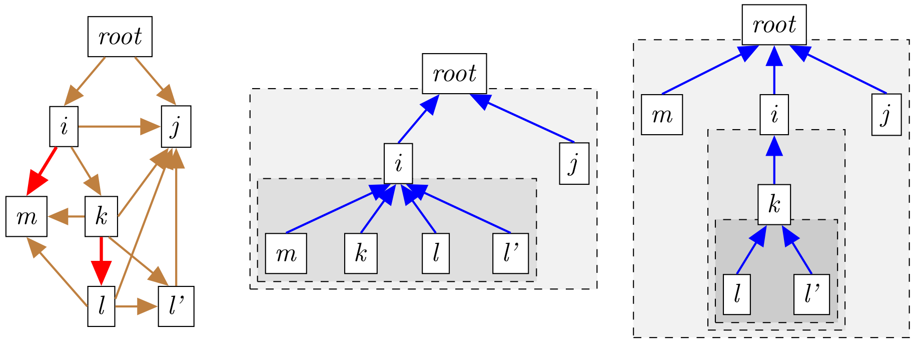
4.4.2 Objective Function

OT cannot use an objective function with one qualifier per partition. Informally, the base modifiers are preference-ranked as

$$\mathsf{rep} > \mathsf{own} > \mathsf{p}$$

but, say, $\langle \mathsf{rep}|\mathsf{rep} \rangle$ should not carry more weight than $\langle \mathsf{rep}|\mathsf{p} \rangle$. The objective function should maximize the number of rep owners *regardless of ownership parameters.*

To illustrate this point, suppose that qualifiers $\langle q_0|q_1 \rangle$ were ordered lexicographically based on the ranking of base modifiers, and consider Figure 4.4. A variable



(a) Object graph (b) OT tree for Γ_1 (c) OT tree for Γ_2

Figure 4.4: Ownership trees resulting from typings Γ_1 and Γ_2. Edges $i \to m$ and $k \to l$ (shown in red in the object graph) cannot be typed with owner rep simultaneously.

roughly corresponds to an edge in the object graph, and therefore, we use typing of edges instead of typing of variables. Edges $i \to m$ and $k \to l$ cannot be typed with owner rep simultaneously, because of the restriction to one ownership parameter. Thus, one valid typing, call it Γ_1, types $root \to i$, $root \to j$ and $i \to m$ as $\langle \mathsf{rep}|\mathsf{rep} \rangle$, $i \to k$ as $\langle \mathsf{rep}|\mathsf{own} \rangle$, and the rest of the edges as either $\langle \mathsf{own}|\_ \rangle$ or $\langle \mathsf{p}|\mathsf{p} \rangle$. Γ_1 gives rise to the ownership tree in Figure 4.4(b); Γ_1 flattens the tree at l and l' — the owner of l and l' is i, even though k dominates both l and l' and we would like to have k as the owner of l and l'. Another valid typing, call it Γ_2, types $root \to i$, $root \to j$ as $\langle \mathsf{rep}|\mathsf{rep} \rangle$, $i \to k$ as $\langle \mathsf{rep}|\mathsf{own} \rangle$, $k \to l$ and $k \to l'$ as $\langle \mathsf{rep}|\mathsf{p} \rangle$, and the rest of the edges as either $\langle \mathsf{own}|\_ \rangle$ or $\langle \mathsf{p}|\mathsf{p} \rangle$. Γ_2 gives rise to the tree in Figure 4.4(c); this tree is better than the tree in Figure 4.4(b) because it has more dominance. Note that lexicographical ordering ranks Γ_1 higher than Γ_2 because it contains 3 $\langle \mathsf{rep}|\mathsf{rep} \rangle$ typings, while Γ_2 contains only 2 $\langle \mathsf{rep}|\mathsf{rep} \rangle$ typings. However, Γ_2 is the better typing, because it contains 5 $\langle \mathsf{rep}|\_ \rangle$ typings, one more than Γ_1, and therefore, it preserves more dominance in the ownership tree than Γ_1.

In OT, the objective function is instantiated as

$$o_{OT}(\Gamma) = (|\Gamma^{-1}(\langle \mathsf{rep}|\_ \rangle)|, |\Gamma^{-1}(\langle \mathsf{own}|\_ \rangle)|, |\Gamma^{-1}(\langle \mathsf{p}|\_ \rangle)|)$$

Here $\Gamma^{-1}(\langle \mathsf{rep}|\_ \rangle)$ is the set of variables typed with owner rep, i.e., typed $\langle \mathsf{rep}|\mathsf{rep} \rangle$,

$\langle \mathsf{rep}|\mathsf{own}\rangle$ or $\langle \mathsf{rep}|\mathsf{p}\rangle$. $\Gamma^{-1}(\langle \mathsf{own}|\_\rangle)$ is the set of variables typed with owner own, and $\Gamma^{-1}(\langle \mathsf{p}|\_\rangle)$ is the set of variables typed with owner p. The primary goal is to maximize the number of variables typed with owner rep (regardless of ownership parameters). Thus, the ranking *maximizes the number of edges in the object graph that are typed* rep , or in other words, the best typing preserves the most dominance (ownership). This is a good proxy for a deep OT ownership tree. o_{OT} does not give rise to such ranking (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle$ and $\langle \mathsf{rep}|\mathsf{p}\rangle$ are equally preferred by o_{OT}). We use lexicographical order over the base modifiers:

$$O_{OT} : \langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle > \langle \mathsf{own}|\mathsf{p}\rangle > \langle \mathsf{p}|\mathsf{p}\rangle$$

O_{OT} preserves the partition ranking (e.g., $\langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle$) and preference-ranks qualifiers within partitions (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle$).

4.4.3 Maximal Typing

There are multiple typings that maximize the objective function o_{OT} for Ownership Types. Consider the following program:

x = new X(); $\boxed{x}$
y = new Y(); $\boxed{y}$
x.f = y;

There are variables x, y, field f, and allocation sites x and y. Typing Γ_1 types the program as follows: $\Gamma_1(\mathsf{x}) = \Gamma_1(x) = \langle \mathsf{rep}|\mathsf{own}\rangle$, $\Gamma_1(\mathsf{y}) = \Gamma_1(y) = \langle \mathsf{rep}|\mathsf{own}\rangle$, and $\Gamma_1(\mathsf{f}) = \langle \mathsf{own}|\mathsf{p}\rangle$. Typing Γ_2 types the program as follows:. $\Gamma_2(\mathsf{x}) = \Gamma_2(x) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, $\Gamma_2(\mathsf{y}) = \Gamma_2(y) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, and $\Gamma_2(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle$. Clearly, $o_{OT}(\Gamma_1) = o_{OT}(\Gamma_2) = (4, 1, 0)$. There are other valid typings that maximize o_{OT} as well. There are nontrivial examples as well.

In addition, the optimality property does not always hold in Ownership Types. As an example, consider the program:

x = new A();
y = new $\langle \boldsymbol{\mathsf{own}}|\boldsymbol{\mathsf{own}}\rangle$ C();
x.f = y;

The application of transfer functions yields $S(\mathsf{x}) = \{\langle\mathsf{rep}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{p}\rangle\}$, $S(\mathsf{f}) = \{\langle\mathsf{own}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{p}\rangle, \langle\mathsf{p}|\mathsf{p}\rangle\}$ and $S(\mathsf{y}) = \{\langle\mathsf{own}|\mathsf{own}\rangle\}$. If we map every variable to the maximal qualifier we have

$$\Gamma(\mathsf{x}) = \langle\mathsf{rep}|\mathsf{own}\rangle,\ \Gamma(\mathsf{f}) = \langle\mathsf{own}|\mathsf{own}\rangle,\ \Gamma(\mathsf{y}) = \langle\mathsf{own}|\mathsf{own}\rangle$$

which fails to type-check because $\langle\mathsf{rep}|\mathsf{own}\rangle \rhd \langle\mathsf{own}|\mathsf{own}\rangle$ equals $\langle\mathsf{rep}|\mathsf{rep}\rangle$, not $\langle\mathsf{own}|\mathsf{own}\rangle$. The set-based solution contains several valid typings. If we chose the maximal value at x, we will have typing

$$\Gamma(\mathsf{x}) = \langle\mathsf{rep}|\mathsf{own}\rangle,\ \Gamma(\mathsf{f}) = \langle\mathsf{p}|\mathsf{p}\rangle,\ \Gamma(\mathsf{y}) = \langle\mathsf{own}|\mathsf{own}\rangle$$

and if we chose the maximal value at f, we will have

$$\Gamma(\mathsf{x}) = \langle\mathsf{own}|\mathsf{own}\rangle,\ \Gamma(\mathsf{f}) = \langle\mathsf{own}|\mathsf{own}\rangle,\ \Gamma(\mathsf{y}) = \langle\mathsf{own}|\mathsf{own}\rangle$$

The set-based solution is valuable for two reasons. First, it restricts the search space significantly. Initially, there are 6 possibilities for each variable and there are n variables, leading to 6^n potential typings. Second, the set-based solution highlights the points of non-determinism where programmer-provided annotations can guide the inference to choose one typing over another. With a small number of programmer-provided annotations, OT inference can scale up to large programs. We explain the process in the remainder of this section.

The points of non-determinism arise at field access and method call statements due to viewpoint adaptation. A statement s is a *conflict* if it does not type-check with the maximal assignment derived from the set-based solution. In the example above, statement $\mathsf{x.f} = \mathsf{y}$ is a conflict, because if we map every variable to the maximal qualifier, the statement fails to type-check. Our approach performs the following incremental process. Given a program P, which may be unannotated or partially annotated, the tool runs the set-based solver, and if there are conflicts, these conflicts are printed. The programmer selects a subset of conflicts (usually the first 1 to 5), and for each conflict, annotates variables. Then the programmer runs

Table 4.1: Inference of Ownership Types for the example in Figure 4.1.

Variable	Initial	Iteration 1	Iteration 2	Iteration 3
top	all	all	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$
d1	all	$\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$
newTop	all	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$
new Link()	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$	$\langle\mathsf{rep}\mid\mathsf{p}\rangle$
s	all	all	all	all
new XStack()	all	all	all	all
x	all	all	all	all
new X()	all	all	all	all
next	all	$\langle\mathsf{own}\mid\mathsf{own}\rangle$, $\langle\mathsf{own}\mid\mathsf{p}\rangle$, $\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{own}\mid\mathsf{p}\rangle$	$\langle\mathsf{own}\mid\mathsf{p}\rangle$
data	all	$\langle\mathsf{own}\mid\mathsf{own}\rangle$, $\langle\mathsf{own}\mid\mathsf{p}\rangle$, $\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$
d2	all	$\langle\mathsf{own}\mid\mathsf{own}\rangle$, $\langle\mathsf{own}\mid\mathsf{p}\rangle$, $\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$	$\langle\mathsf{p}\mid\mathsf{p}\rangle$

the set-based solver again. This process continues until a program P' is reached, where the optimality property holds for P'. The solver computes a maximal typing for P'.

In the above example, the solver prints conflict $\mathsf{x.f} = \mathsf{y}$ and the set-based solution

$$\begin{aligned} S(\mathsf{x}) &= \{\langle\mathsf{rep}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{p}\rangle\} \\ S(\mathsf{f}) &= \{\langle\mathsf{own}|\mathsf{own}\rangle, \langle\mathsf{own}|\mathsf{p}\rangle, \langle\mathsf{p}|\mathsf{p}\rangle\} \\ S(\mathsf{y}) &= \{\langle\mathsf{own}|\mathsf{own}\rangle\} \end{aligned}$$

If the programmer chooses to annotate x with $\langle\mathsf{rep}|\mathsf{own}\rangle$, this results in typing

$$\Gamma(\mathsf{x}) = \langle\mathsf{rep}|\mathsf{own}\rangle,\ \Gamma(\mathsf{f}) = \langle\mathsf{p}|\mathsf{p}\rangle,\ \Gamma(\mathsf{y}) = \langle\mathsf{own}|\mathsf{own}\rangle$$

and if he/she chooses to annotate f with $\langle\mathsf{own}|\mathsf{own}\rangle$ this results in typing

$$\Gamma(\mathsf{x}) = \langle\mathsf{own}|\mathsf{own}\rangle,\ \Gamma(\mathsf{f}) = \langle\mathsf{own}|\mathsf{own}\rangle,\ \Gamma(\mathsf{y}) = \langle\mathsf{own}|\mathsf{own}\rangle$$

4.5 Inference Example

Table 4.1 shows the computation of the set-based solution for the example program in Figure 4.1. Note that this computation assumes annotation $\langle\mathsf{rep}|\mathsf{p}\rangle$ at allocation site new Link(); given this annotation, the optimality property holds, and the set-based solution computes the maximal typing for the program.

CHAPTER 5
Universe Types

In Chapter 4 we have discussed the classic Ownership Types which enforce the *owner-as-dominator* encapsulation discipline. Other ownership type systems enforce the *owner-as-modifier* discipline: an object can be referenced by any other object, but access paths that do not go through its owner cannot modify it. That is, an object *o* can only be modified by its owner, or its peers (i.e. the objects that have the same owner as *o*).

Universe Types (UT) [8, 20], is a lightweight ownership type system that optionally enforces the *owner-as-modifier* encapsulation discipline. Universe Types distinguishes *readwrite* references and *readonly* references, and the owner-as-modifier discipline is only enforced on readwrite references [8]. Owners only control modifications of owned objects, but not read access.

In this chapter, we instantiate the inference framework with Universe Types.

5.1 Type Qualifiers and Subtyping Relation

There are three source-level qualifiers in UT:

- peer: an object that is referenced by a peer reference x is part of the same representation as the current object. In other words, the two objects have the same owner.

- rep: an object that is referenced by a rep reference x is part of the current (i.e., this) object's representation. In other words, the current object is the *owner* of the object referenced by x.

- any: the any qualifier does not provide any information about the ownership of the object.

Portions of this chapter previously appeared as: W. Huang *et al.*, "Inference and checking of object ownership," in *Proc. European Conf. Object-Oriented Programming*, Beijing, China, 2012, pp. 181–206.

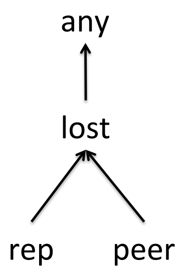


Figure 5.1: Subtying Hierarchy of Universe Types.

The formalization of Universe Types uses the qualifier lost to express that the result of viewpoint adaptation cannot be expressed statically, that is, a type declaration enforces an ownership constraint, but the constraint is not expressible from the current viewpoint. Qualifier lost is used only internally and users cannot annotate references as lost.

The qualifiers form the following subtyping relation:

$$\mathsf{rep} <: \mathsf{lost} \qquad \mathsf{peer} <: \mathsf{lost} \qquad \mathsf{lost} <: \mathsf{any}$$

that is, qualifiers peer and rep are incomparable to each other and are subtypes of lost, and all qualifiers are below any. The subtyping hierarchy is shown in Figure 5.1.

Figure 5.2 shows a doubly-linked list implementation from [8] annotated with UT. Class Node contains two peer references: prev and next, and an any reference elem. first at line 7 is declared as rep denoting that the object referred by first is owned by the LinkedList instance. The capture method at line 12 is trying to assign some foreign node to the first field. But the assignment at line 13 is illegal, because first and n have different owners: first's owner is the current LinkedList instance, while n's owner is the owner of the current LinkedList instance, as n is of type peer.

The assignment at line 18 is also illegal because it violates the owner-as-modifier discipline by directly modifying the internal representation of another list l. This violation can be detected statically. The owner of l.first is l, because first is of type rep. According to the owner-as-modifier discipline, an object can only be modified by its owner or its peers. The current LinkedList instance is neither the owner nor peers of l.first, thus this assignment is detected as illegal. This is more clear in the

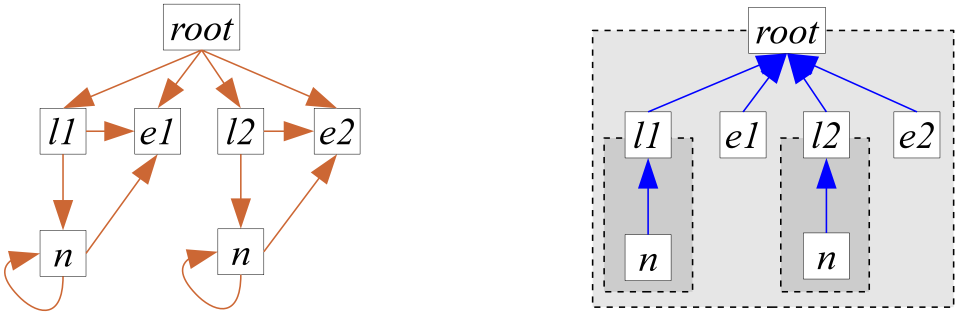
```
class Node {
  peer Node prev;
  peer Node next;
  any Object elem;
}
class LinkedList {
  rep Node first;
  LinkedList(any Object e) {
    first = new rep Node(); n
    first.elem = e;
  }
  void capture(peer Node n) {
    first = n; // illegal
  }
  void exchangeFirst(peer LinkedList l) {
    any Object tmp = first.elem;
    first.elem = l.first.elem;
    l.first.elem = tmp; // illegal
  }
  void static main(String[] args) {
    Object e1 = new Object(); e1
    LinkedList l1 = new LinkedList(e1); l1
    Object e2 = new Object(); e2
    LinkedList l2 = new LinkedList(e2); l2
    l2.exchangeFirst(l1);
  }
}
```

Figure 5.2: Annotated doubly-linked list.

resulting object graph and ownership tree in Figure 5.3. The n in the shadow of $l1$ is owned by $l1$, but $l2$ is neither the owner nor the peer of this n. Hence, modification of this n is illegal in $l2$.

5.2 Viewpoint Adaptation

Viewpoint adaptation in Universe Types encodes the relation between the current this object and its transitively reachable objects.



(a) Object graph

(b) Ownership tree of LinkedList in UT

Figure 5.3: Object graph (left) and ownership tree (right) for LinkedList.

5.2.1 Viewpoint Adaptation Operation

Similarly to Ownership Types, viewpoint adaptation is applied only when the receiver variable is not this. Therefore, we reuse the self qualifier as in Ownership Types. Viewpoint adaptation operation of Universe Types is defined as follows:

$$\begin{array}{llllll}
\mathsf{peer} & \rhd & \mathsf{peer} & = & \mathsf{peer} & \\
\mathsf{rep} & \rhd & \mathsf{peer} & = & \mathsf{rep} & \\
\_ & \rhd & \mathsf{any} & = & \mathsf{any} & \\
\mathsf{self} & \rhd & q & = & q & \\
q & \rhd & q' & = & \mathsf{lost} & \text{otherwise}
\end{array}$$

5.2.2 Context of Adaptation

Similar to OT, the context of adaptation function distinguishes the cases when the receiver is this at field access and method call:

$$\mathcal{C}(\mathsf{y.f} = \mathsf{x}) \quad = \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}(\mathsf{x} = \mathsf{y.f}) \quad = \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}(\mathsf{x} = \mathsf{y.m}^{i}(\mathsf{z})) \quad = \begin{cases} \mathsf{self} & \text{if } \mathsf{y} = \mathsf{this} \\ q_{\mathsf{y}} & \text{otherwise} \end{cases}$$

For example, consider $\mathsf{y.f} = \mathsf{x}$ where $\mathsf{y} \neq \mathsf{this}$ and the context of adaptation is q_{y}. If y is rep, then the current object is the owner of the y object. If the type of f is peer, then the y object and field f object are peers. Therefore, the current object is the owner of the f object, which is expressed by the fact that the type of f, adapted from the point of view of y's rep, is rep.

5.2.3 Additional Constraints

Universe Types impose additional constraints, beyond the standard subtyping and viewpoint adaptation constraints. In (TNEW), the newly created object needs to be created in a concrete ownership context and therefore needs peer or rep as ownership qualifiers. In (TWRITE), the adapted field type cannot be lost, and in (TCALL), the adapted formal parameter type cannot be lost. The function $\mathcal{B}$ is defined as follows:

$$\begin{array}{lcl}
\mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) & = & \{q \neq \mathsf{any}\} \\
\mathcal{B}(\mathsf{y.f} = \mathsf{x}) & = & \{\underline{q_{\mathsf{y}} \neq \mathsf{any}},\ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \neq \mathsf{lost}\} \\
\mathcal{B}(\mathsf{x} = \mathsf{y.m}^{i}(\mathsf{z})) & = & \text{let } \mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \text{ in} \\
 & & \quad \text{if } \mathit{impure}(\mathsf{m}) \text{ then } \{\underline{q_{\mathsf{y}} \neq \mathsf{any}},\ q_{\mathsf{y}} \rhd q_p \neq \mathsf{lost}\} \\
 & & \quad \text{else} \quad \{q_{\mathsf{y}} \rhd q_p \neq \mathsf{lost}\}
\end{array}$$

The $\mathcal{B}$ sets for (TASSIGN) and (TREAD) are all empty; these rules do not impose additional constraints.

The underlined constraints above enforce the owner-as-modifier encapsulation discipline — they disallow modifications in statically unknown contexts. The receiver cannot be any in (TWRITE) or in (TCALL) if the method is impure, that is, if the method might have nonlocal side effects. We use our method purity inference tool, described in Section 3.7 and in [15]. Note that, in contrast to other formalizations [17], we do not need to forbid lost as receiver, because our syntax here is in "named form" and the programmer cannot explicitly write lost.

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \qquad \mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) = \{q \neq \mathsf{any}\}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\textsc{tnew}) \qquad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}} \qquad \mathcal{B}(\mathsf{y.f} = \mathsf{x}) = \{q_{\mathsf{y}} \neq \mathsf{any},\ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \neq \mathsf{lost}\}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}\ (\textsc{twrite})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \qquad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \qquad q_{\mathsf{x}} <: q_{\mathsf{f}}}{\Gamma \vdash \mathsf{this.f} = \mathsf{x}}\ (\textsc{twritethis}) \qquad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \qquad q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{this.f}}\ (\textsc{treadthis})$$

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \to q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} <: q_{\mathsf{y}} \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} <: q_{\mathsf{y}} \rhd q_p \quad q_{\mathsf{y}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) = \{q_{\mathsf{y}} \rhd q_p \neq \mathsf{lost} \text{ and } \mathit{impure}(\mathsf{m}) \Rightarrow q_{\mathsf{y}} \neq \mathsf{any}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}\ (\textsc{tcall})$$

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \to q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} <: q_{\mathsf{this}} \quad q_{\mathsf{z}} <: q_p \quad q_{\mathsf{ret}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{this.m}^i(\mathsf{z})}\ (\textsc{tcallthis})$$

Figure 5.4: Instantiated Typing Rules for Universe Types. Again, we show separate typing rules when the receiver is this.

5.3 Instantiated Typing Rules

The instantiated typing rules for Universe Types are shown in Figure 5.4. Note that the default type for the implicit parameter this is peer. Again, we separate the typing rules (TWRITE), (TREAD), and (TCALL) when the receiver is this for readability. The instantiated typing rules still fit into the framework.

5.4 Type Inference

In this section, we first define the initial set mapping S_0 and the objective function o_{UT} to leverage the inference framework for type inference of Universe Types. Then we discuss the maximal typing for Universe Types.

5.4.1 Initial Mapping

The mapping S_0 is initialized as follows. Programmer-annotated variables are initialized to the singleton set of the provided type. The implicit parameter this is initialized $S(\mathsf{this}) = \{\mathsf{self}\}$. Library variables are initialized to $\{\mathsf{peer}\}$. All other variables are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\mathsf{rep}, \mathsf{peer}, \mathsf{any}\}$.

5.4.2 Objective Function

In Universe Types, the objective function is instantiated as

$$o_{UT}(T) = (|T^{-1}(\mathsf{any})|, |T^{-1}(\mathsf{rep})|, |T^{-1}(\mathsf{peer})|)$$

The partitioning and ordering is

$$\{\mathsf{any}\} > \{\mathsf{rep}\} > \{\mathsf{peer}\}$$

Each qualifier falls in its own partition. This means, informally, that we prefer any over rep and peer, and rep over peer. More formally, the partitioning and ordering gives rise to a preference ranking O_{UT} over all qualifiers:

$$O_{UT} : \mathsf{any} > \mathsf{rep} > \mathsf{peer}$$

Function o_{UT} gives a natural ranking over the set of valid typings for UT. In fact, the maximal (i.e., best) typing according to the above ranking, *maximizes the number of allocation sites typed* rep , which is a good proxy for a deep UT ownership tree.

5.4.3 Maximal Typing

For Universe Types, the optimality property holds for unannotated programs, which means that the unique maximal typing maximizes that objective function o_{UT}.

The maximal typing provably always type-checks. We show through case analysis that for each statement s, after the application of the SOLVECONSTRAINT(c) function for each constraint c generated from statement s, s type-checks with the maximal typing, i.e. all constraints generated from s are satisfiable:

(TASSIGN) Consider $\mathsf{x} = \mathsf{y}$ which generates constraint $c : \mathsf{y} <: \mathsf{x}$. We must show that after the application of SOLVECONSTRAINT(c), c is satisfiable with $max(S'(\mathsf{x}))$ and $max(S'(\mathsf{y}))$.

- If $max(S'(\mathsf{x})) = \mathsf{any}$, c is satisfiable with any value for $max(S'(\mathsf{y}))$.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$. Thus, any is not in $S'(\mathsf{x})$, and therefore any cannot be in $S'(\mathsf{y})$. $max(S'(\mathsf{y}))$ cannot be peer; this contradicts the assumption that $max(S'(\mathsf{x})) = \mathsf{rep}$ (rep would have been removed from x's set). Thus, $max(S'(\mathsf{y})) = \mathsf{rep}$ and c is satisfiable.
- Suppose now that $max(S'(\mathsf{x})) = \mathsf{peer}$. The only possible value for $max(S'(\mathsf{y}))$ is peer and c again is satisfiable.

(TNEW) is shown exactly the same way.

(TREAD) Consider $\mathsf{x} = \mathsf{y.f}$ which generates constraint c: $\mathsf{y} \rhd \mathsf{f} <: \mathsf{x}$. We must show that after the application of SOLVECONSTRAINT(c), c is satisfiable with $max(S'(\mathsf{x}))$, $max(S'(\mathsf{f}))$ and $max(S'(\mathsf{y}))$.

- If $max(S'(\mathsf{x})) = \mathsf{any}$ this is clearly true.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$; then $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be rep.
- Finally, when $max(S'(\mathsf{x})) = \mathsf{peer}$, one can easily see that $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be peer as well.

(TWRITE) and (TCALL) are analogous; they are omitted for brevity. We have implemented an independent type checker, which verifies the inferred solution.

```
 1  class XStack {
 2    any Link top;
 3    XStack() {
 4      top = null;
 5    }
 6    void push(any X d1) {
 7      rep Link newTop;
 8      newTop = new rep Link(); $\boxed{l}$
 9      newTop.init(d1);
10      newTop.next = top;
11      top = newTop;
12    }
13    void main(String[] arg) {
14      rep XStack s;
15      s = new rep XStack(); $\boxed{s}$
16      any X x = new rep X(); $\boxed{x}$
17      s.push(x);
18    }
19  }
20  class Link {
21    any Link next;
22    any X data;
23    void init(any X d2) {
24      next = null;
25      data = d2;
26    }
27  }
```

Figure 5.5: Annotated XStack with Universe Types. The boxed italic letters denote object allocation sites.

5.5 Inference Example

Figure 5.5 shows a program annotated with Universe types. Variable newTop at line 7 and the Link object l are typed rep, meaning that the XStack object is the owner of the Link object. References top (line 2) and next (line 21) are any because they are never used to modify the object that they refer to. References d1 (line 6) and d2 (line 23) are any as well, as they are never used to modify the object they refer to.

Table 5.6 illustrates the computation of the set-based solution for UT for

Figure 5.6: Inference of Universe Types for the example in Figure 5.5.

Variable	Initial	Iteration 1	Iteration 2
top	all	all	all
d1	all	any, peer	any, peer
newTop	all	rep, peer	rep, peer
new Link()	all	rep, peer	rep, peer
s	all	rep, peer	rep, peer
new XStack()	all	rep, peer	rep, peer
x	all	all	all
new X()	all	rep, peer	rep, peer
next	all	any, peer	any, peer
data	all	any, peer	any, peer
d2	all	any, peer	any, peer

the example in Figure 5.5. Consider statement s.push(x) at line 17. Initially, $S(\mathsf{s}) = S(\mathsf{x}) = S(\mathsf{d1}) = \{\mathsf{any}, \mathsf{rep}, \mathsf{peer}\}$. In iteration 1, the transfer function for s.push(x) removes any from $S(\mathsf{s})$ because push is impure. It also removes rep from $S(\mathsf{d1})$ because $q \rhd \mathsf{rep} = \mathsf{lost}$ which the type rule for (TCALL) forbids. See Table 5.6. Choosing the maximal type from each set gives us $T(\mathsf{s}) = \mathsf{rep}$, $T(\mathsf{x}) = \mathsf{any}$, and $T(\mathsf{d1}) = \mathsf{any}$, which type-checks with the rule for (TCALL).

CHAPTER 6
Information Flow Systems

In this chapter, we consider information type systems. The general structure of these systems is as follows. The universe of type qualifiers is

$$U = \{\mathsf{neg}, \mathsf{poly}, \mathsf{pos}\}$$

with subtyping hierarchy

$$\mathsf{neg} <: \mathsf{poly} <: \mathsf{pos}$$

Here neg is the "negative" qualifier and pos is the "positive" qualifier. The goal of the type system is to ensure that there is no flow from a "positive" variable x to a "negative" variable y. poly is a polymorphic qualifier, which is interpreted as pos in some contexts, and as neg in other contexts.

The best examples are confidentiality and integrity information flow systems. A confidentiality system instantiates neg to public and pos to secret. The goal of the system is to ensure that there is no flow from secret variables (i.e., *sources*) to public variables (i.e., *sinks*), or in other words, to disallow flow from secret variables to public variables. Intuitively, it is safe to assign a public variable to a secret one, but it is not safe to assign a secret variable to a public one; hence the direction of the subtyping relation: public $<:$ secret. A confidentiality system prevents sensitive data from flowing into untrusted sinks. Other examples of information flow systems include EnerJ [11], ReIm [15], and more.

In this chapter, we first instantiate the inference framework with SFlow/Integrity (Section 6.1), for detecting information flow vulnerabilities in Java web applications [41]. Information flow vulnerabilities are one of the most common security problems for web applications according to OWASP [42]. Examples of information flow vulnerabilities include SQL injection, cross-site scripting (XXS), HTTP response

Portions of this chapter previously appeared as: W. Huang *et al.*, "Type-based taint analysis for Java web application," in *Int. Conf. Fundamental Approaches to Software Engineering*, Grenoble, France, 2014, pp. 140–154.

splitting, path traversal and command injection [43].

We then instantiate the inference framework with SFlow/Confidentiality (Section 6.2), for detecting privacy leaks in Android apps. Android users are often subjected to privacy leaks, e.g. the apps may obtain and abuse sensitive data such the phone identifier, without the user's consent.

6.1 SFlow/Integrity for Java Web Applications

Let us begin with a motivating example shown in Figure 6.1 (adapted from [43]).

```
HttpServletRequest request = ...;
Statement stat = ...;
String user = request.getParameter(”user”);
StringBuffer sb = ...;
sb.append(”SELECT * FROM Users WHERE name = ”);
sb.append(user);
String query = sb.toString();
stat.executeQuery(query);
```

Figure 6.1: SQL injection example.

In this example, the user parameter of the HTTP request obtained through request.getParameter(“user”) and stored in variable user, which is later appended to an SQL query string and sent to a database for execution: stat.executeQuery(query). At a first glance, this code snippet is unremarkable. However, if a malicious end-user supplies the user parameter with the value of “John OR 1 = 1”, the unauthorized end-user can gain access to the information of all other users, because the WHERE clause always evaluates to true.

Static taint analysis detects information flow vulnerabilities. It automatically detects flow from untrusted *sources* to security-sensitive *sinks*. In the example in Figure 6.1, the return value of HttpServletRequest.getParameter() is a source, and the parameter p of Statement.executeQuery(String p) is a sink.

Research on static taint analysis for Java web applications has largely focused on dataflow and points-to-based approaches [43, 44, 45, 46, 47]. One issue with these approaches is that they usually rely on context-sensitive points-to analysis, which is expensive and non-modular (i.e., it requires a whole program). Arguably the toughest issue is dealing with reflection, libraries (JDK and third-party), and frameworks (Struts, Spring, Hibernate, etc.), features notoriously difficult for dataflow and points-to analysis and yet ubiquitous in Java web applications.

In this section, we instantiate the inference framework with SFlow/Integrity, a context-sensitive type system for secure information flow, and leverage the inference framework for detecting information flow violations in Java web applications.

6.1.1 Type Qualifiers and Subtying Relation

There are three basic type qualifiers in SFlow/Integrity: tainted, safe, and poly.

- tainted: A variable x is tainted, if there is flow from a source to x. Sources, e.g., the return value of ServletRequest.getParameter(), are annotated as tainted.
- safe: A variable x is safe if there is flow from x to a sensitive sink. Sinks, e.g., the parameter p of Statement.executeQuery(String p), are annotated as safe.
- poly: The poly qualifier expresses context sensitivity. poly is interpreted as tainted in some invocation contexts and as safe in other contexts. This is analogous to the polyread qualifier in ReIm (see Chapter 3).

SFlow/Integrity disallows flow from tainted sources to safe sinks. Therefore, the subtyping relation between these type qualifiers is defined as:

$$\mathsf{safe} <: \mathsf{poly} <: \mathsf{tainted}$$[2]

Thus, assigning a safe variable to a tainted one is allowed:

```
safe int s = ...;
tainted int t = s;
```

but assigning a tainted variable to a safe one is disallowed:

```
tainted int t = ...;
safe int s = t; // type error!
```

[2] Note that this is the desired subtyping. Unfortunately, this subtyping is not always safe, as we discuss in detail in Section 6.1.3

In the SQL injection example in Figure 6.1, the return value of getParameter() is annotated as tainted, and the parameter of executeQuery(String p) is annotated as safe, as they are a source and a sink, respectively. The other variables are tainted as shown in Figure 6.2. Since it is not allowed to assign the tainted query to the safe parameter of executeQuery(String p), statement 8 does not type-check, resulting in a type error. The type error signals an information flow violation.

```
1  HttpServletRequest request = ...;
2  Statement stat = ...;
3  tainted String user = request.getParameter("user");
4  tainted StringBuffer sb = ...;    // it includes the tainted user
5  sb.append("SELECT * FROM Users WHERE name = ");
6  sb.append(user);
7  tainted String query = sb.toString();
8  stat.executeQuery(query);         // type error!
```

Figure 6.2: SQL injection example with SFlow/Integrity typing.

6.1.2 Viewpoint Adaptation

As in ReIm, viewpoint adaptation in SFlow/Integrity encodes context sensitivity, which is crucial to the typing precision of SFlow/Integrity.

6.1.2.1 Context Sensitivity

In the context-insensitive typing in Figure 6.2, methods append and toString must be typed as follows:

tainted StringBuffer append(**tainted** StringBuffer this, **tainted** String s) {...}
tainted String toString(**tainted** StringBuffer this) {...}

Such context-insensitive typing is imprecise, because it types the return value of toString as tainted. Consider the example in Figure 6.3. query at line 7 is not tainted by any input, but it is *typed* tainted because the return value of toString is of type tainted. Therefore, the program is rejected, even though it is safe.

SFlow/Integrity achieves context sensitivity by making use of a polymorphic type qualifier, poly, and *viewpoint adaptation*. The poly qualifiers must be interpreted

```
1 String user = request.getParameter("user");
2 StringBuffer sb1 = ...; StringBuffer sb2 = ...;
3 sb1.append("SELECT * FROM Users WHERE name = ");
4 sb2.append("SELECT * FROM Users WHERE name = ");
5 sb1.append(user);
6 sb2.append("John");
7 String query = sb2.toString();
8 stat.executeQuery(query); // type error in context-insensitive typing
```

Figure 6.3: Context sensitivity example.

according to invocation context. Intuitively, the role of *viewpoint adaptation* (which we elaborate upon shortly), is to interpret the poly qualifiers according to the invocation context. In Figure 6.3, poly is interpreted as tainted at call sb1.append(user), and as safe at call sb2.append("John"). As a result, the tainted argument in the call through sb1 does not propagate to sb2; thus, query at line 7 is typed safe, and the type error at line 8 is avoided.

6.1.2.2 Viewpoint Adaptation Operation

The viewpoint adaptation operation in SFlow/Integrity is defined as follows:

$$\begin{array}{cclcl} \_ & \rhd & \mathsf{tainted} & = & \mathsf{tainted} \\ \_ & \rhd & \mathsf{safe} & = & \mathsf{safe} \\ q & \rhd & \mathsf{poly} & = & q \end{array}$$

The underscore denotes a “don’t care” value. Qualifiers tainted and safe do not depend on the viewpoint (context). Qualifier poly depends on the viewpoint; in fact, it adapts to that viewpoint (context).

6.1.2.3 Context of Adaptation

The context of adaptation for SFlow/Integrity is always the *receiver* at field access or method call. The function $\mathcal{C}$ is defined as follows:

$$\begin{aligned} \mathcal{C}(\mathsf{y.f} = \mathsf{x}) &= q_{\mathsf{y}} \\ \mathcal{C}(\mathsf{x} = \mathsf{y.f}) &= q_{\mathsf{y}} \\ \mathcal{C}(\mathsf{x} = \mathsf{y.m}^{i}(\mathsf{z})) &= q_{\mathsf{y}} \end{aligned}$$

For example, the type of a poly field f is interpreted in the context of the receiver y according to the viewpoint adaptation operation. If the receiver y is tainted, then y.f is tainted. If the receiver y is safe, then y.f is safe.

6.1.3 Composition with Reference Immutability

In order to improve the typing precision, we compose SFlow/Integrity with reference immutability by using additional constraints.

The reader has likely noticed that subtyping safe <: poly <: tainted is not always sound. Suppose the field f of class A is poly in the following example:

```
tainted B tf = ...;
safe A s = ...;
tainted A t = s;   // because of safe <: tainted
t.f = tf;          // t.f is tainted
safe B sf = s.f;   // s.f is safe, unsafe flow!
```

The program type-checks, but the tainted variable tf flows to safe variable sf. This is the known problem of subtyping in the presence of mutable references, also known as the issue with Java's covariant arrays [48].

The standard solution is to disallow subtyping for references [11, 49]. For example, EnerJ [11] defines two sets of qualifiers: precise <: poly <: approx for simple types, and Precise, Poly, Approx for references. While subtyping is allowed for simple types, it is disallowed for references. This solution demands two sets of qualifiers, safe <: poly <: tainted for simple types (e.g., `int,char`), and Safe, Poly, Tainted for reference types. While subtyping is allowed for simple types, it is disallowed for reference types.

Unfortunately, disallowing subtyping for reference types leads to imprecision, i.e., the type system rejects valid programs. It amounts to using *equality constraints* as opposed to subtyping constraints, and thus, propagating safe and tainted qualifiers bi-directionally, resulting in often unnecessary propagation. Disallowing subtyping is in some sense analogous to using unification constraints as opposed to subset constraints in points-to analysis. It is well-known that Steensgaard's points-to analysis [50], which uses unification (i.e., equality) constraints, is substantially less precise than Andersen's points-to analysis [51], which uses subset constraints.

To illustrate the problem, consider a micro benchmark from Stanford's Securibench-micro from `http://suif.stanford.edu/~livshits/work/securibench-micro/` (Retrieved on April 29, 2014):

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String name = req.getParameter(FIELD_NAME);
    String str = "abc";
    name = str;
    PrintWriter writer = resp.getWriter();
    writer.println(str);
}
```

where the return type of HttpServletRequest.getParameter is tainted, and the parameter of PrintWriter.println is safe. There is no dangerous flow here, as the tainted name never flows to println. If we disallowed subtyping for references however, name = str would have required that name is safe and type-checking would have failed, even though the program is safe.

We propose a solution using reference immutability, which allows limited subtyping and improves precision. Subtyping is safe when the reference on the left-hand-side of the assignment is an *immutable reference*, that is, the state of the referenced object, including its transitively reachable state, *cannot be mutated through this reference.*

We compose SFlow/Integrity with ReIm (see Chapter 3). We first run type inference for ReIm and obtain ReIm types for all variables. If the ReIm type of the left-hand-side of an assignment is readonly, i.e., it is guaranteed that this left-hand-side is immutable, we use a subtyping constraint in SFlow. Otherwise, i.e., if the ReIm type is not readonly, we use an equality constraint, by adding an additional subtyping constraint in the opposite direction in the additional constraints $\mathcal{B}$. The additional constraints $\mathcal{B}$ are defined as follows:

$$\begin{array}{lcl}
\mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q\} \\
\mathcal{B}(\mathsf{x} = \mathsf{y}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}}\} \\
\mathcal{B}(\mathsf{y.f} = \mathsf{x}) & = & \{reim(\mathsf{f}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}\} \\
\mathcal{B}(\mathsf{x} = \mathsf{y.f}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}\} \\
\mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) & = & \{reim(\mathsf{this}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{this}} <: q_{\mathsf{y}}, \\
 & & \ reim(\mathsf{p}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{p}} <: q_{\mathsf{z}}, \\
 & & \ reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{ret}}\}
\end{array}$$

where $reim$ is a function that retrieves the inferred ReIm type for a variable and we have instantiated the context of adaptation to q_{y}. For example, at (TREAD) $\mathsf{x} = \mathsf{y.f}$, if x is readonly, we use constraint $q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}$; otherwise, we add the additional constraint $\mathcal{B}(\mathsf{x} = \mathsf{y.f}) = \{q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}\}$. Note that $reim(\mathsf{x}) \neq \mathsf{readonly}$ is true and q_c is q_{y}. The equality constraint is enforced by two subtyping constraints $q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}$ and $q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}$.

6.1.4 Instantiated Typing Rules

The instantiated typing rules for SFlow/Integrity are shown in Figure 6.4. Let us return to the example in Figure 6.3 and consider the typing rule (TCALL). In the context-sensitive SFlow/Integrity, the polymorphic method append is typed as follows:

poly StringBuffer append(**poly** StringBuffer this, **poly** String s) {...}

Let sb1 be typed tainted. The call at line 5, namely sb1.append(user), accounts for the following constraint according to (TCALL):

$$\mathsf{user} <: \mathsf{sb1} \rhd \mathsf{s} \ \equiv\ \mathsf{user} <: \mathsf{sb1} \rhd \mathsf{poly} \ \equiv\ \mathsf{user} <: \mathsf{sb1}$$

Since user and sb1 are tainted, the call at line 5 type-checks. Now let sb2 be typed safe. The call at line 6, sb2.append("John"), accounts for constraint according to (TCALL):

$$\text{“John”} <: \mathsf{sb2} \rhd \mathsf{s} \ \equiv\ \text{“John”} <: \mathsf{sb2} \rhd \mathsf{poly} \ \equiv\ \text{“John”} <: \mathsf{sb2}$$

Since string constant "John" and sb2 are both safe, this type-checks as well. In the first context of invocation of append we interpreted poly s as tainted, while in the

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\textsc{tnew})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad typeof(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ q_{\mathsf{x}} \ <: \ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \\ \mathcal{B}(\mathsf{y.f} = \mathsf{x}) = \{reim(\mathsf{f}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}\}\end{array}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}\ (\textsc{twrite})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad typeof(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \ <: \ q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.f}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\begin{array}{c}typeof(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} \ <: \ q_{\mathsf{y}} \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} \ <: \ q_{\mathsf{y}} \rhd q_p \quad q_{\mathsf{y}} \rhd q_{\mathsf{ret}} \ <: \ q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) = \{reim(\mathsf{this}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{this}} <: q_{\mathsf{y}}, \\ \qquad reim(\mathsf{p}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_p <: q_{\mathsf{z}}, \\ \qquad reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{ret}}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}\ (\textsc{tcall})$$

Figure 6.4: Instantiated Typing Rules for SFlow/Integrity.

second context, we interpreted it as safe.

As it is evident from these typing rules, we consider only explicit flows (i.e., data dependences). To the best of our knowledge, all effective static taint analyses [52, 53, 43, 44, 46, 45, 47] forgo implicit flows.

6.1.5 Type Inference

In this section, we first define the initial set mapping and the objective function for SFlow/Integrity. Unfortunately, the optimality property does not hold for SFlow/Integrity, which means the maximal typing extracted from the set-based solution does not type-check. Therefore, we extend the set-based solver with the computation of the *method summary constraints* to further remove infeasible qualifiers from the set-based mapping and help arrive at a valid typing.

6.1.5.1 Initial Mapping

The mapping S_0 is initialized as follows. Programmer-annotated variables are initialized to the singleton set of the provided type. In SFlow/Integrity, programmers annotate the sources and the sinks as $\mathsf{tainted}$ and safe, respectively. For Java web applications, sources and sinks are usually from Java libraries. Therefore, once the programmers have inserted annotations into libraries, these annotated libraries can be reused in the inference for other web applications. Fields f are initialized to $S(\mathsf{f}) = \{\mathsf{tainted}, \mathsf{poly}\}$. This is because a $\mathsf{tainted}$ object should not contain safe information in its fields — it is not clear what the meaning of such an object would be. All other variables are initialized to the maximal set of qualifiers, i.e. $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

6.1.5.2 Objective Function

In SFlow/Integrity, the objective function is instantiated as

$$o_{SFlow/Integrity}(T) = (|T^{-1}(\mathsf{tainted})|, |T^{-1}(\mathsf{poly})|, |T^{-1}(\mathsf{safe})|)$$

The partitioning and ordering is

$$\{\mathsf{tainted}\} > \{\mathsf{poly}\} > \{\mathsf{safe}\}$$

Each qualifier falls in its own partition. This means that we prefer $\mathsf{tainted}$ over poly and safe, and poly over safe. The maximal typing according to the above ranking, maximizes the number of variables typed $\mathsf{tainted}$.

```
class A {
  String f;
  String get()
  {return this.f;}            this ▷ f <: ret
}
A y = ...;
PrintWriter writer = ...;
String x = y.get();           y <: y ▷ this    y ▷ ret <: x
writer.print(x);              x <: writer ▷ safe
```

Figure 6.5: An example illustrating the computation of the method summary constraints. The frame box beside each statement shows the corresponding constraints the statement generates.

6.1.5.3 Method Summary Constraints

Unfortunately, the optimality property does not always hold in SFlow/Integrity. Consider the example in Figure 6.5. The inference solves the generated constraints and yields the set-based solution S: $S(\mathsf{x}) = \{\mathsf{safe}\}$, $S(\mathsf{y}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$, $S(\mathsf{this}) = \{\mathsf{poly}, \mathsf{safe}\}$, $S(\mathsf{ret}) = \{\mathsf{poly}, \mathsf{safe}\}$, and $S(\mathsf{f}) = \{\mathsf{poly}\}$. If we pick the maximal qualifier from each set, the maximal typing is

$$\begin{array}{lcl} \Gamma(\mathsf{x}) & = & \mathsf{safe} \\ \Gamma(\mathsf{y}) & = & \mathsf{tainted} \\ \Gamma(\mathsf{this}) & = & \mathsf{poly} \\ \Gamma(\mathsf{ret}) & = & \mathsf{poly} \\ \Gamma(\mathsf{f}) & = & \mathsf{poly} \end{array}$$

This maximal typing fails to type-check because the constraint $\mathsf{y} \rhd \mathsf{ret} <: \mathsf{x}$ is not satisfied:

$$\mathsf{y} \rhd \mathsf{ret} \equiv \mathsf{tainted} \rhd \mathsf{poly} = \mathsf{tainted}$$

and $\mathsf{tainted}$ is not a subtype of $\Gamma(\mathsf{x}) = \mathsf{safe}$.

The set-based solver removes many infeasible qualifiers and in many cases, it discovers type errors. In our experience, the set-based solver, which is quadratic in the worst case and linear in practice, discovers the vast majority of type errors, and therefore it is useful on its own. Unfortunately, when the set-based solver terminates without type errors, it is unclear if a valid typing exists or not, and therefore, there is no guarantee of safety. The question is, how do we extract a valid typing, or conversely, show that a valid typing does not exist?

The idea is to compute what we call *method summary constraints*, which remove additional qualifiers from the set-based solution. These constraints reflect the relations (subtyping or equality) between formal parameters (including this) and return values (ret). Such references are usually "connected" indirectly, e.g. this and ret can be connected through two constraints $\mathsf{this} <: \mathsf{x}$ and $\mathsf{x} <: \mathsf{ret}$. Note that intuitively, the subtyping relation reflects flow: there is flow from this to x, there is flow from x to ret, and due to transitivity, there is flow from this to ret.

Once we have computed the relations between formal parameters and return values of a method m, we connect the actual arguments to the left hand sides of the call assignment at calls to m. The computation of method summary constraints is presented in Figure 6.6.

Let us return to the example in Figure 6.5. Case 2 in Figure 6.6 creates $\mathsf{this} <: \mathsf{ret}$. This entails $\mathsf{y} \rhd \mathsf{this} <: \mathsf{y} \rhd \mathsf{ret}$ since viewpoint adaptation preserves subtyping [54]. Case 3 combines this with constraints $\mathsf{y} <: \mathsf{y} \rhd \mathsf{this}$ and $\mathsf{y} \rhd \mathsf{ret} <: \mathsf{x}$, yielding a new constraint $\mathsf{y} <: \mathsf{x}$. Because $\mathsf{tainted}$ and poly are not subtypes of safe, SolveConstraint removes them from $S(\mathsf{y})$, and $S(\mathsf{y})$ becomes $\{\mathsf{safe}\}$. The set-based solution is updated to $S(\mathsf{x}) = \{\mathsf{safe}\}$, $S(\mathsf{y}) = \{\mathsf{safe}\}$, $S(\mathsf{this}) = \{\mathsf{poly}, \mathsf{safe}\}$, $S(\mathsf{ret}) = \{\mathsf{poly}, \mathsf{safe}\}$, and $S(\mathsf{f}) = \{\mathsf{poly}\}$. We obtain a valid maximal typing:

$$\begin{array}{lcl} \Gamma(\mathsf{x}) & = & \mathsf{safe} \\ \Gamma(\mathsf{y}) & = & \mathsf{safe} \\ \Gamma(\mathsf{this}) & = & \mathsf{poly} \\ \Gamma(\mathsf{ret}) & = & \mathsf{poly} \\ \Gamma(\mathsf{f}) & = & \mathsf{poly} \end{array}$$

In our experiments (see Section 7.3), the maximal typing always type-checks, except for 2 constraints in one benchmark, jugjobs. It is a theorem that even if it does not type-check, the program is still safe, i.e., there is no flow from sources to

```
1: procedure RunSolver
2:     repeat
3:         for each $c$ in $C$ do
4:             SolveConstraint($c$)
5:             if $c$ is $q_\mathsf{x} <: q_\mathsf{y} \rhd q_\mathsf{f}$ and $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ then                ▷ Case 1
6:                 Add $q_\mathsf{x} <: q_\mathsf{y}$ into $C$
7:             else if $c$ is $q_\mathsf{x} \rhd q_\mathsf{f} <: q_\mathsf{y}$ and $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ then           ▷ Case 2
8:                 Add $q_\mathsf{x} <: q_\mathsf{y}$ into $C$
9:             else if $c$ is $q_\mathsf{x} <: q_\mathsf{y}$ then                                ▷ Case 3
10:                for each $q_\mathsf{y} <: q_\mathsf{z}$ in $C$ do add $q_\mathsf{x} <: q_\mathsf{z}$ to $C$ end for
11:                for each $q_\mathsf{w} <: q_\mathsf{x}$ in $C$ do add $q_\mathsf{w} <: q_\mathsf{y}$ to $C$ end for
12:                for each $q_\mathsf{w} <: q_\mathsf{a} \rhd q_\mathsf{x}$ and $q_\mathsf{a} \rhd q_\mathsf{y} <: q_\mathsf{z}$ in $C$ do     ▷ Case 4
13:                    Add $q_\mathsf{w} <: q_\mathsf{z}$ to $C$
14:                end for
15:            end if
16:        end for
17:    until $S$ remains unchanged
18: end procedure
```

Figure 6.6: Computation of method summary constraints. C is the set of constraints, it is initialized to the set of constraints for program statements (recall that each equality constraint is written as two subtyping constraints). S is initialized to the result of the set-based solver. Cases 1 and 2 add $q_\mathsf{x} <: q_\mathsf{y}$ into C because $q_\mathsf{y} \rhd \mathsf{poly}$ always yields q_y. Case 3 adds constraints due to transitivity; this case discovers constraints from formals to return values. Case 4 adds constraints between actual(s) and left-hand-side(s) at calls: if there are constraints $q_\mathsf{w} <: q_\mathsf{a} \rhd q_\mathsf{x}$ (flow from actual to formal) and $q_\mathsf{a} \rhd q_\mathsf{y} <: q_\mathsf{z}$ (flow from return value to left-hand-side), and also $q_\mathsf{x} <: q_\mathsf{y}$ (flow from formal to return value, usually discovered by Case 3), Case 4 adds $q_\mathsf{w} <: q_\mathsf{z}$. Note that line 4 calls SolveConstraint(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until S stays unchanged.

sinks. We confirmed this for the 2 constraints in jugjobs.

Complexity After extending the set-based solver with method summary constraints, the inference in Figure 6.6 reaches the *fixpoint* (when S stays unchanged) in $O(n^3)$ time, where n is the size of the program. There are at most $O(3n)$ iterations of the outer loop (line 2), because in each iteration at least one of $O(n)$ references is updated to refer to a smaller set of qualifiers, and each set has at most 3 qualifiers.

```
 1  void doGet(A this, ServletRequest request, ServletResponse response) {
 2    StringBuffer buf = ...;
 3    this.foo(buf,buf,request,response);     buf = this_doGet ▷ b1     S(buf) = {tainted}
 4  }                                          buf <: this_doGet ▷ b2    S(b2) = {tainted, poly}
 5  void foo(A this, StringBuffer b1, StringBuffer b2,
 6        ServletRequest req, ServletResponse resp) {
 7    String url = req.getParameter("url");    req ▷ tainted <: url      S(url) = {tainted}
 8    b1.append(url);                          url <: b1 ▷ poly          S(b1) = {tainted}
 9    String str = b2.toString();              b2 ▷ poly <: str          S(str) = {tainted, poly}
10    PrintWriter writer = resp.getWriter();
11    writer.print(str);                       str <: writer ▷ safe      TYPE ERROR!
12  }
```

Figure 6.7: Aliasing5 example from Stanford SecuriBench Micro. The frame box beside each statement shows the corresponding constraints the statement generates. The oval boxes show propagation during the set-based solution. The constraint at 7 forces url to be tainted, and the constraint at 8 forces b1 to be tainted. The constraint at 3 forces buf to be tainted and the one at 4 forces b2 to be tainted or poly (i.e., the set-based solver removes safe from b2's set). The constraint at 9 then forces str to be tainted or poly. There is a TYPE ERROR at writer.print(str).

The inner loop (line 3) iterates over at most $O(n^2)$ constraints, because in the worst case every two references can form a constraint, resulting in $O(n^2)$ constraints. Altogether, we have worst-case complexity of $O(n^3)$. Although at first glance lines 10-13 (Cases 3-4) appear to contribute $O(n) * O(n^2) * O(3n)$, a closer look reveals they contribute only $O(n) * O(n^2)$, or $O(n^3)$ (this is because lines 10-13 run only when a new constraint $q_{\mathsf{x}} <: q_{\mathsf{y}}$ is discovered, and there are at most $O(n^2)$ such new constraints).

6.1.6 Inference Example

To demonstrate the power of the type system and inference analysis, we elaborate on two examples that have posed challenges for previous taint analyses [43, 46].

The Aliasing5 example from Ben Livshits' Stanford SecuriBench Micro benchmarks[3] in Figure 6.7 illustrates the handling of aliasing. foo is safe when b1 and b2 refer to distinct StringBuffer objects. However, when b1 and b2 are aliased, foo creates dangerous flow from source req.getParameter to a sink, the parameter of PrintWriter.print. Note that the constraint at line 3 is an equality constraint: b1 is mutated at b1.append(url), ReIm infers b1 as mutable, and hence the equality constraint. The set-based solver reports a type error at statement 11; the constraint at 11 is unsatisfiable as it requires that str is safe, which contradicts the finding that str is $\{\mathsf{tainted}, \mathsf{poly}\}$.

The second example, shown in Figure 6.8 illustrates the handling of context sensitivity. There are two instances of DataSource, one that holds a tainted string in its f field, and another one that holds a safe string. The code is safe because s2, which flows to the sensitive sink, is read from the “safe” DataSource object. A context-insensitive taint analysis would merge the flows through setUrl and getUrl across the two different instances of DataSource, and report a spurious warning.

Figure 6.8 illustrates our solution. The inferred typing types class DataSource as polymorphic. The poly types are instantiated to tainted for object ds1 and to safe for object ds2.

As illustrated, the analysis handles naturally these difficult idioms. The handling of DataSource can be interpreted as object sensitivity [55]: essentially, the analysis processes polymorphic setUrl and getUrl separately for object contexts ds1 and ds2, just as standard object-sensitive analysis does.

6.1.7 Web Application-Specific Features

Reflection, libraries (standard and third-party) and frameworks (e.g., Struts, Spring, Hibernate) are the bane of static taint analysis. Yet they are ubiquitous in Java web applications. In addition, mapping data structures such as Propterties, HashMap, etc. are key to connecting flows between the front-end (e.g. JSP pages) and the back-end (e.g. Servlets) in Java web applications. The type-based approach we espouse, handles these features safely and effortlessly.

[3] `http://suif.stanford.edu/~livshits/work/securibench-micro/` (Retrieved on April 29, 2014).

		Constraint	Set-based Solution
1	class DataSource {		
2	String f;		
3	void setUrl(String url) {		$S(\text{this}_{\text{setUrl}}) = \{\text{tainted}, \textbf{poly}\}$
4	this.f = url;	$\text{url} <: \text{this}_{\text{setUrl}} \rhd \text{f}$	$S(\text{url}) = \{\text{tainted}, \textbf{poly}\}$
5	}	$\text{url} <: \text{this}_{\text{setUrl}}$	
6	String getUrl() {		$S(\text{this}_{\text{getUrl}}) = \{\text{tainted}, \textbf{poly}\}$
7	return this.f;	$\text{this}_{\text{getUrl}} \rhd \text{f} <: \text{ret}_{\text{getUrl}}$	$\text{S(f)} = \{\textbf{poly}\}$
8	}	$\text{this}_{\text{getUrl}} <: \text{ret}_{\text{getUrl}}$	$S(\text{ret}_{\text{getUrl}}) = \{\textbf{poly}, \text{safe}\}$
9	}		
10	String tUrl = req.getParameter(..);	$\text{req} \rhd \textbf{tainted} <: \text{tUrl}$	$\text{S(tUrl)} = \{\textbf{tainted}\}$
11	DataSource ds1 = new DataSource();		
12	ds1.setUrl(tUrl);	$\text{tUrl} <: \text{ds1} \rhd \text{url}$	
		$\text{ds1} = \text{ds1} \rhd \text{this}_{\text{setUrl}}$	$S(\text{ds1}) = \{\textbf{tainted}, \text{poly}, \text{safe}\}$
14	String sUrl = "http://localhost/";		$S(\text{sUrl}) = \{\text{tainted}, \text{poly}, \textbf{safe}\}$
15	DataSource ds2 = new DataSource();		
16	ds2.setUrl(sUrl);	$\text{sUrl} <: \text{ds2} \rhd \text{url}$	
		$\text{ds2} = \text{ds2} \rhd \text{this}_{\text{setUrl}}$	$S(\text{ds2}) = \{\text{tainted}, \text{poly}, \textbf{safe}\}$
18	String s1 = ds1.getUrl();	$\text{ds1} <: \text{ds1} \rhd \text{this}_{\text{getUrl}}$	
		$\text{ds1} \rhd \text{ret}_{\text{getUrl}} <: \text{s1}$	$S(\text{s1}) = \{\textbf{tainted}, \text{poly}, \text{safe}\}$
		$\text{ds1} <: \text{s1}$	
21	String s2 = ds2.getUrl();	$\text{ds2} <: \text{ds2} \rhd \text{this}_{\text{getUrl}}$	
		$\text{ds2} \rhd \text{ret}_{\text{getUrl}} <: \text{s2}$	
		$\text{ds2} <: \text{s2}$	
24	writer.println(s2);	$\text{s2} <: \text{writer} \rhd \textbf{safe}$	$S(\text{s2}) = \{\textbf{safe}\}$

Figure 6.8: The DataSource example due to Ben Livshits [43]. The frame box beside each statement shows the generated constraints correspondingly. The bold red frame boxes show the constraints generated by the algorithm in Figure 6.6. The oval boxes show the set-based solution, where overstruck qualifiers are eliminated by the the algorithm in Figure 6.6. The bold qualifiers are the final maximal typing. It type-checks.

6.1.7.1 Reflective Object Creation

Use of reflective object creation in web applications is widespread. Ignoring it, as some static analyses do, renders a static analysis useless. Consider the use of newInstance():

```
X x = (X) Class.forName("someInput").newInstance();
x.f = a;     // a is tainted, comes from source
y = x;
b = y.f;     // b is safe, flows to sink
```

If a points-to-based static analysis fails to handle newInstance(), the points-to sets of x and y will be empty, and the flow from a to b will be missed. On the other hand, handling of reflective object creation is difficult, expensive and often unsound.

We handle reflective object creation with newInstance() safely and effortlessly. The key is that SFlow tracks dependences between variables through subtyping, which *obviates the need to abstract heap objects.* It can be shown that, roughly speaking, if x flows to y, then x $<:$ y holds. In the above example, x $<:$ y holds and subsequently a $<:$ b holds. The type inference reports a type error caused by the flow from tainted a to safe b.

6.1.7.2 Libraries

Our inference analysis is modular. Thus, it can analyze any given set of classes L. If there is an unknown callee in L, e.g. a library method whose source code is unavailable, the analysis assumes typing poly, poly → poly for the callee. This typing conservatively propagates tainted arguments to the receiver and left-hand-side of the call assignment. Similarly, it propagates a safe left-hand-side to the receiver and arguments at the call. E.g., String.toUpperCase() is typed as

poly String toUpperCase(**poly** String this)

At call s2 = s1.toUpperCase() we have constraint s1 $\rhd$ poly $<:$ s2 or equivalently s1 $<:$ s2. Thus, a tainted s1 propagates to s2, and a safe s2 propagates to s1.

We apply the poly, poly → poly typing to all methods in the standard library, third-party libraries (e.g., apache-tomcat, xalan) and frameworks, with several exceptions described in the next section.

6.1.7.3 Frameworks

Most Java web applications are built on top of one or more *frameworks* such as Struts, Spring, Hibernate, and etc. The problem with these frameworks is twofold. First, they contain "hidden" sources and sinks, i.e., sources and sinks deep in framework code that affect the public API. For example, Hibernate (version 2.1) contains a public method Session.find(String s), where s flows to query at sink prepareStatement(query). This happens deep in the code of Hibernate. We run a version of our inference analysis and "lift" such hidden sources and sinks to the return values and parameters of the public methods they affect. In the above example, Session.find() is typed as

poly List find(**poly** Session this, **safe** String s)

Callers to find() in application code must handle the argument of find() as safe, otherwise it may lead to an SQL injection vulnerability as described by Livshits and Lam [43]. To the best of our knowledge, no other taint analysis attempts to "lift" these "hidden" sources and sinks in the frameworks.

Second, these frameworks rely heavily on reflection and callbacks, which must be handled in the analysis. These are notorious issues for dataflow and points-to based analysis, which usually relies on reachability analysis. Our type-based analysis handles these features with the method overriding constraints.

As an illustrating example, Struts defines framework classes ActionForm and Action and method Action.execute(ActionForm form). The application built on top of Struts defines numerous xxxForm classes extending ActionForm, and numerous xxxAction classes extending Action. Framework code performs the following (roughly):

1. Action a = (Action) Class.forName("inputClass").newInstance(); a instantiates one user-defined xxxAction class.

2. ActionForm f = (ActionForm) Class.forName("inputForm").newInstance(); similarly, this instantiates one user-defined xxxForm class.

3. Framework populates the xxxForm object with *tainted* values from sources.

4. Framework calls a.execute(f), a callback to user-defined xxxAction.execute.

In our type-based analysis Action.execute() is typed as

execute(**poly** Action this, **tainted** ActionForm form)

```
 1  class BlojsomServlet {
 2    public static final String AUTHOR = "BLOJSOM_AUTHOR";
 3    public void doGet(HttpServletRequest req, HttpServletResponse resp) {
 4      String inAuthor = req.getParameter(‘‘author’’); // tainted source
 5      req.setAttribute(BLOJSOM_AUTHOR, inAuthor);
 6    }
 7  }
 8  class html_dcomments_jsp {
 9    public void _jspService(HttpServletRequest req, HttpServletResponse resp) {
10      String outAuthor = (String) req.getAttribute(BlojsomServlet.AUTHOR);
11      PrintWriter out = ...;
12      out.print(outAuthor);     // safe sink
13    }
14  }
```

Figure 6.9: Imprecision caused by mapping data structures.

The method overriding constraints (recall Section 2.2.3) propagate tainted to the form parameter of each execute method in user-defined subclasses. As a result, all values retrieved through get methods from forms in user code are tainted, which accurately reflects that the xxxForm object is populated with tainted values.

6.1.7.4 Mapping Data Structures

We special-case global mapping data structures Properties from the java.util package, and ServletRequest and HttpSession from the javax.servlet package. In order to illustrate the problem, consider the example in Figure 6.9 refactored from benchmark blojsom. At line 6, the tainted inAuthor is put into the mapping of req. Then it is retrieved at line 13 through req.getAttribute() and printed to the client page. The parameter of PrintWriter.print() is a safe sink according to [43]. Therefore, there is unsafe flow from req.getParameter() to out.print().

If outAuthor = req.getAttribute(...) were handled according to the typing rules in Figure 6.4, the safe outAuthor would cause req to be safe, and safe would propagate to all calls on receiver req, not only to the call req.setAttribute(...,inAuthor).

Therefore, we special-case set$*$ and get$*$ methods for such mapping data structures, similarly to Sridharan et al. [45]. If the key of the set$*$ method call set(key, value) is a constant, the inference simply creates the equality constraint key = value. Similarly, if the key of get$*$ method call x = get(key) is a constant, the set-based solver creates constraint x = key. For the example in Figure 6.9, the set-based solver enforces BlojsomServlet.BLOJSOM_AUTHOR = inAuthor at line 5 and outAuthor = BlojsomServlet.BLOJSOM_AUTHOR at line 10. Thus, inAuthor and outAuthor are connected and outAuthor is typed as tainted. The unsafe information flow is detected because there is a type error when passing tainted outAuthor to the safe parameter of out.print().

6.2 SFlow/Confidentiality for Android apps

In this section, we consider the confidentiality system SFlow/Confidentiality for detecting privacy leaks in Android apps.

Android is the most popular platform on mobile devices. As of November 2013, Android has reached 81% share of the global smartphone market [56]. Android's success is partly due to the enormous number of applications available at the Google Play Store, as well as other third-party app stores. In the meantime, Android users are often subjected to malicious behaviors of the apps they have installed. Sensitive data such as phone identifier, location information, SMS messages, etc. can be obtained and abused by malicious apps. The *Mobile Thread Report* from F-Secure shows that in the third quarter of 2013, 97% of the mobile malware targeted Android [57].

Android's coarse-grained permission-based security model is not sufficient to regulate access to sensitive data. An app declares permissions such as ACCESS_FINE_LOCATION and INTERNET statically, and the user grants such permissions at installation time. However, the app can abuse those permissions, e.g. the app can send the location information to an untrusted targeted advertising service. Android lacks the fine-grained permission control that would allow to specify *where* sensitive data can flow.

Many researchers have tackled taint analysis for Android. Dynamic analyses such as Google Bouncer [58], TaintDroid [59], DroidScope [60], CopperDroid [61], and Aurasium [62] instrument the app bytecode and/or use customized execution environment to monitor the transition of sensitive data. Unfortunately, dynamic analysis slows execution and typically lacks code coverage.

Research on static taint analysis for Android has largely focused on dataflow and points-to-based approaches [47, 63, 64, 65, 66]. One issue with such approaches is that they usually rely on context-sensitive points-to analysis, which is expensive and typically requires a whole program analysis. Unfortunately, Android apps are not whole programs. They are "open" in the sense that they have no main method; they run within the Android framework by implementing callback methods which are called by the framework, corresponding to different events or states of the lifecycle. Therefore, without precisely modeling the app's lifecycle and accurately discovering the app's entry points, such static approaches would be unsound and ineffective in practice.

This section presents the dual confidentiality system SFlow/Confidentiality and its application for detecting privacy leaks in Android apps. The SFlow/Confidentiality type system is similar to SFlow/Integrity, except that SFlow/Confidentiality has different names for type qualifiers and uses a different context of adaptation to improve typing precision.

6.2.1 Motivating Example

The example shown in Figure 6.10 is refactored from one of our benchmarks, "Backgrounds HD Wallpapers" version 2.0.1 from the Google Play Store. The WallpapersMain activity first obtains the device identifier by calling the getDeviceId method and stores it into a field deviceId when it is created (onCreate). Then it appends the deviceId into a search URL url, which is sent to a content server in the navigate method. Finally, the navigate method is called in callback method onActivityResult, resulting in a privacy leak.

This example poses several challenges to traditional points-to-based dataflow analyses. First, unlike Java programs, Android apps do not have a single entry point. Instead, each callback method is a potential entry point as it could be called by the Android framework. In WallpapersMain, both onCreate and onActivityResult are callback methods that are implicitly called by the Android framework. An Android

```
 1 public class WallpapersMain extends Activity {
 2   private String BASE_URL, deviceId;
 3   private int pageNum, catId;
 4   private DisplayMetrics metrics;
 5   private WebView browser1;
 6   protected void onCreate(Bundle b) {
 7     start();
 8   }
 9   protected void onActivityResult(int rq, int rs, Intent i) {
10     navigate();
11   }
12   private void start() {
13     BASE_URL = "getWallpapers_Android2/";
14     TelephonyManager mgr =
15      (TelephonyManager) this.getSystemService("phone");
16     deviceId = mgr.getDeviceId(); // source
17   }
18   private void navigate() {
19     String str = BASE_URL + pageNum + "/" + catId + "/" + deviceId + "/"
           + metrics.widthPixels + "/" + metrics.heightPixels;
20     browser1.loadUrl(str); // sink
21   }
22 }
```

Figure 6.10: WallpapersMain **leaks the phone identifier (the source at line 16) to a content server (the sink at line 20) in a URL.**

app consists of a number of *components*, each of which can be instantiated and run within the Android framework.

The Android app defines its own behaviors at different states of the component lifecyle by overriding pre-defined callback methods. Multiple entry points challenge points-to-based static analyses, which usually require whole program analysis and precise call graphs. Second, control flow is interrupted by callbacks from Android. In the WallpapersMain example, control flow from onCreate to onActivityResult must be captured in order to detect the leak.

6.2.2 Type Qualifiers and Subtyping Relation

SFlow/Confidentiality is the dual confidentiality system of SFlow/Integrity and there are three type qualifiers in SFlow/Confidentiality: secret, public, and poly.

- secret: A variable x is secret, if there is flow from a sensitive source to x. In the WallpapersMain example, the return value of TelephonyManager.getDeviceId is typed as secret. This is is the positive qualifier, equivalent to tainted in SFlow/Integrity.

- public: A variable x is public if there is flow from x to an untrusted sink. For example, the parameter url of WebView.loadUrl(String url) is a public sink. This is the negative qualifier, equivalent to safe in SFlow/Integrity.

- poly: The poly qualifier expresses context sensitivity. poly is interpreted as secret in some contexts and as public in other contexts. This is the same as poly in SFlow/Integrity.

The subtyping relation is

public <: poly <: secret

Similarly to SFlow/Integrity, this is the desired subtyping relation. SFlow/Confidentiality also composes with ReIm to improve the typing precision. It is allowed to assign a public variable to a secret one:

```
public String s = ...;
secret String t = s;
```

However, it is not allowed to assign a secret variable to a public one:

```
secret String t = ...;
public String s = t; // type error!
```

In the WallpapersMain example, the return value of getDeviceId is typed as secret and the url parameter of loadUrl is typed as public, as they are a source and a sink, respectively. The field deviceId is typed as secret and so is the local variable str since it contains the value of deviceId. Because it is not allowed to assign a secret str to the public parameter url of loadUrl, Statement 20 does not type-check, resulting in a type error. This type error indicates a privacy leak.

Once the sources and sinks are given, type qualifiers are inferred automatically using the inference framework. If there is a valid typing, then there is no flow from a source to a sink. Otherwise, i.e., if there is no valid typing, the tool reports type errors, signaling potential privacy leaks.

6.2.3 Viewpoint Adaptation

The viewpoint adaptation function in SFlow/Confidentiality is defined as follows:

$$\begin{array}{lcl} \_ \rhd \mathsf{secret} & = & \mathsf{secret} \\ \_ \rhd \mathsf{public} & = & \mathsf{public} \\ q \rhd \mathsf{poly} & = & q \end{array}$$

For a field access, SFlow/Confidentiality uses the receiver as the context of adaptation, which is the same as SFlow/Integrity:

$$\begin{array}{lcl} \mathcal{C}(\mathsf{y.f} = \mathsf{x}) & = & q_{\mathsf{y}} \\ \mathcal{C}(\mathsf{x} = \mathsf{y.f}) & = & q_{\mathsf{y}} \end{array}$$

For a method call, SFlow/Confidentiality improves the typing precision by using the callsite q^i rather than the receiver as the context of adaptation.

$$\mathcal{C}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) = q^i$$

This improvement allows SFlow/Confidentiality to accept more valid programs, as we will elaborate shortly in Section 6.2.5.

6.2.4 Additional Constraints

SFlow/Confidentiality also composes with ReIm and $\mathcal{B}$ is defined as follows:

$$\begin{array}{lcl} \mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q\} \\ \mathcal{B}(\mathsf{x} = \mathsf{y}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}}\} \\ \mathcal{B}(\mathsf{y.f} = \mathsf{x}) & = & \{reim(\mathsf{f}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}\} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.f}) & = & \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}\} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) & = & \{reim(\mathsf{this}) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_{\mathsf{this}} <: q_{\mathsf{y}}, \\ & & reim(\mathsf{p}) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_{\mathsf{p}} <: q_{\mathsf{z}}, \\ & & reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q^i \rhd q_{\mathsf{ret}}\} \end{array}$$

$$\textsc{(tnew)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{new}\ q\ \mathsf{C}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}$$

$$\textsc{(tassign)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

$$\textsc{(twrite)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ q_{\mathsf{x}} \ <: \ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \\ \mathcal{B}(\mathsf{y.f} = \mathsf{x}) = \{reim(\mathsf{f}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}\}\end{array}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

$$\textsc{(tread)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{y}} \rhd q_{\mathsf{f}} \ <: \ q_{\mathsf{x}} \\ \mathcal{B}(\mathsf{x} = \mathsf{y.f}) = \{reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}\}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

$$\textsc{(tcall)}$$

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} \ <: \ q^i \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} \ <: \ q^i \rhd q_p \quad q^i \rhd q_{\mathsf{ret}} \ <: \ q_{\mathsf{x}} \\ \begin{array}{rl}\mathcal{B}(\mathsf{x} = \mathsf{y.m}^i(\mathsf{z})) = \{ & reim(\mathsf{this}) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_{\mathsf{this}} <: q_{\mathsf{y}}, \\ & reim(\mathsf{p}) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_{\mathsf{p}} <: q_{\mathsf{z}}, \\ & reim(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{x}} <: q^i \rhd q_{\mathsf{ret}}\}\end{array}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}$$

Figure 6.11: Instantiated Typing Rules for SFlow/Confidentiality.

where $reim$ is a function that retrieves the inferred ReIm type for a variable.

6.2.5 Instantiated Typing Rules

The Instantiated typing rules are shown in Figure 6.11.

Note that SFlow/Confidentiality improves the typing precision by using the callsite q^i rather than the receiver as the context of adaptation. Consider the example in Figure 6.12, where method id is typed as follows:

```
 1  class Util {
 2    poly String id(secret Util this, poly String p) {
 3      return p;
 4    }
 5  }
 6  ...
 7  Util y = new Util();
 8  secret String src = ...;
 9  public String sink = ...;
10  secret String srcId = y.id(src);
11  public String sinkId = y.id(sink);
```

Figure 6.12: id **example.**

$$\mathsf{poly\ String\ id(secret\ Util\ this,\ poly\ String\ p)}$$

which enables context sensitivity because id can take as input a secret String as well as a public one.

If we used the receiver y as the adaptation context, (TCALL) generates the following constraints at callsite 10:

$$\mathsf{y} <: \mathsf{y} \triangleright \mathsf{secret} \quad \mathsf{src} <: \mathsf{y} \triangleright \mathsf{poly} \quad \mathsf{y} \triangleright \mathsf{poly} <: \mathsf{srcId}$$

Because $\mathsf{src} = \mathsf{secret}$, y must be secret. However, y being secret does not satisfy the constraints at callsite 11:

$$\mathsf{y} <: \mathsf{y} \triangleright \mathsf{secret} \quad \mathsf{sink} <: \mathsf{y} \triangleright \mathsf{poly} \quad \mathsf{y} \triangleright \mathsf{poly} <: \mathsf{sinkId}$$

where both sink and sinkId are public. This is because $\mathsf{y} \triangleright \mathsf{poly} = \mathsf{secret}$ is not a subtype of $\mathsf{sinkId} = \mathsf{public}$. As a result, this program would be rejected by SFlow/Integrity.

In contrast, SFlow/Integrity overcomes this imprecision. The callsite context q^i is a value that is not important, except that it should exist. q^i can be any of $\{\mathsf{secret}, \mathsf{poly}, \mathsf{public}\}$. In Figure 6.12, SFlow/Integrity creates the following constraints at callsite 10:

$$\mathsf{y} <: q^{10} \triangleright \mathsf{secret} \quad \mathsf{src} <: q^{10} \triangleright \mathsf{poly} \quad q^{10} \triangleright \mathsf{poly} <: \mathsf{secret}$$

$q^{10} = \mathsf{secret}$ satisfies the above constraints. SFlow/Integrity creates the following constraints at callsite 11:

$$\mathsf{y} <: q^{11} \rhd \mathsf{secret} \quad \mathsf{sink} <: q^{11} \rhd \mathsf{poly} \quad q^{11} \rhd \mathsf{poly} <: \mathsf{public}$$

$q^{11} = \mathsf{public}$ satisfies the above constraints. Therefore, SFlow/Confidentiality accepts this program and improves the typing precision by using the callsite as the context of adaptation.

6.2.6 Inference Example

The type inference of SFlow/Confidentiality is similar to SFlow/Integrity. We refer users to the discussion of the type inference for SFlow/Integrity in Section 6.1.5. We present an inference example for SFlow/Confidentiality instead.

Let us consider the FieldSensitivity2 example refactored from DroidBench [47] in Figure 6.13. The return of TelephonyManager.getSimSerialNumber (line 10) is a source and the parameter msg of SmsManager.sendTextMessage (line 16) is a sink. The serial number of the SIM card is obtained and stored into a Data object. Later, it is retrieved from the Data object and sent out through an SMS message without user consent. We illustrate the inference in Figure 6.13 as follows. 16 forces sg to be $\{\mathsf{public}\}$, then 14 forces $\mathsf{ret}_{\mathsf{get}}$ to be $\{\mathsf{poly}, \mathsf{public}\}$ and then 3 forces $\mathsf{this}_{\mathsf{get}}$ to be $\{\mathsf{poly}, \mathsf{public}\}$ and secret to be $\{\mathsf{poly}\}$. 10 forces sim to be $\{\mathsf{secret}\}$, which in turn forces the parameters p and $\mathsf{this}_{\mathsf{set}}$ to be $\{\mathsf{secret}, \mathsf{poly}\}$. There are no type errors in the initial set-based solution. The red frame box in the fourth column (New constraints) shows the computed method summary constraints. Since field secret is poly, constraint $\mathsf{this}_{\mathsf{get}} \rhd \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}$ leads to method summary constraint $\mathsf{this}_{\mathsf{get}} <: \mathsf{ret}_{\mathsf{get}}$, which in turn leads to $\mathsf{dt} <: \mathsf{sg}$ due to the call at 14. Similarly, $\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \rhd \mathsf{secret}$ leads to $\mathsf{p} <: \mathsf{this}_{\mathsf{set}}$, which in turn leads to $\mathsf{sim} <: \mathsf{dt}$ due to the call at 11. Since sim is $\{\mathsf{secret}\}$ and sg is $\{\mathsf{public}\}$, these constraints cause a TYPE ERROR, detecting the leak.

		Constraints	Set-based solution	New constraints
1	public class Data {			
2	String secret;		$S(\text{secret}) = \{\text{poly}\}$	
3	String get(Data this) {return this.secret;}	$\text{this}_{\text{get}} \rhd \text{secret} <: \text{ret}_{\text{get}}$	$S(\text{ret}_{\text{get}}) = \{\text{poly}, \text{public}\}$	$\text{this}_{\text{get}} <: \text{ret}_{\text{get}}$
4	void set(Data this, String p){this.secret = p;}	$\text{p} <: \text{this}_{\text{set}} \rhd \text{secret}$	$S(\text{p}) = \{\text{secret}, \text{poly}\}$	$\text{p} <: \text{this}_{\text{secret}}$
5	}			
6	public class FieldSensitivity2 extends Activity {			
7	protected void onCreate(Bundle b) {			
8	Data dt = new Data();		$S(\text{dt}) = \{\text{secret}, \text{poly}, \text{public}\}$	
9	TelephonyManager tm = (TelephonyManager) getSystemService("phone");			
10	String sim = tm.getSimSerialNumber();	$q^{10} \rhd \textbf{secret} <: \text{sim}$	$S(\text{sim}) = \{\text{secret}\}$	
11	dt.set(sim);	$\text{sim} <: q^{11} \rhd \text{p}$		$\text{sim} <: \text{dt}$
12		$\text{dt} = q^{11} \rhd \text{this}_{\text{set}}$	$S(\text{this}_{\text{set}}) = \{\text{secret}, \text{poly}\}$	
13	SmsManager sms = SmsManager.getDefault();			
14	String sg = dt.get();	$\text{dt} <: q^{14} \rhd \text{this}_{\text{get}}$	$S(\text{this}_{\text{get}}) = \{\text{poly}, \text{public}\}$	
15		$q^{14} \rhd \text{ret}_{\text{get}} <: \text{sg}$		$\text{dt} <: \text{sg}$ TYPE ERROR!
16	sms.sendTextMessage("+123",null,sg,null,null);	$\text{sg} <: q^{16} \rhd \textbf{public}$	$S(\text{sg}) = \{\text{public}\}$	
17	}			
18	}			

Figure 6.13: FieldSensitivity2 example refactored from DroidBench. The frame box beside each statement shows the corresponding constraints the statement generates. We omitted uninteresting constraints. The oval boxes show propagation during the set-based solution.

6.2.7 Android-Specific Features

In this section, we discuss our techniques for handling Android-specific features, including libraries, multiple entry points and callbacks, and inter-component communication.

6.2.7.1 Libraries

Libraries are ubiquitous in Android apps. An effective analysis should keep track of flows through library method calls. Unfortunately, analyzing the Android library is a significant challenge. Computing public summaries for the Android library is an open problem (to the best of our knowledge). Analyzing library calls on-demand, i.e., using some form of reachability analysis faces challenges due to callbacks and reflection, which are pervasive in Android. The most popular solution appears to be manual summaries for common library methods [65, 47], which is clearly unsatisfying.

The inference inserts annotations (type qualifiers) into the Android library for sources (e.g. location access, phone state, contacts) and for sinks (e.g., internet access, storage access) by using the Stub Generation Tool and the Annotation File Utility from the Checker Framework [18]. The inference *uses conservative defaults for all unknown library methods.* For any unanalyzed library method m, it assumes the typing poly, poly → poly. This typing conservatively propagates a secret receiver/argument to the left-hand side of the call assignment. Similarly, it propagates a public left-hand-side to the receiver/arguments. Consider the following code snippet:

```
public class MyListener implements LocationListener {
  @Override
  public void onLocationChanged(Location loc){//source
    double lat = loc.getLatitude();
    Log.d("History", "Latitude: " + lat); // sink
  }
}
```

LocationListener.onLocationChanged(secret Location l) is a callback method. Parameter l is a secret source that must propagate through the overriding method MyListener.onLocationChanged(Location loc). The method overriding constraints

(Section 2.2.3) lead to:

$$\mathit{typeof}(\mathsf{MyListener.onLocationChanged(Location\ loc)})$$
$$<:$$
$$\mathit{typeof}(\mathsf{LocationListener.onLocationChanged(secret\ Location\ l)})$$

This entails $\mathsf{l} <: \mathsf{loc}$, forcing loc to be secret as well.

The inference assumes that library method Location.getLatitude() is typed as follows:

poly double getLatitude(poly Location this)

and creates the following constraints at Statement 4:

$$\mathsf{loc} <: q^4 \rhd \mathsf{poly} \quad q^4 \rhd \mathsf{poly} <: \mathsf{lat}$$

Because loc is secret, the callsite context q^4 is inferred as secret. Consequently, lat is inferred as secret as well, which leads to a type error at Statement 5 where a public argument is required (Here the parameter msg of Log.d(String tag, String msg) is a public sink.) Hence, the privacy leak is captured.

We apply these conservative defaults to the Java and Android libraries. We can apply these defaults to any third-party library we do not wish to analyze.

6.2.7.2 Multiple Entry Points and Callbacks

Multiple entry points and the ubiquitous use of callbacks in Android apps cause difficulty for traditional points-to based static analysis. The Android app is not a closed program. Instead, it runs within the Android framework, which implicitly creates objects of the user-defined classes and calls user-defined methods in the app through callbacks.

The inference is type-based and modular. Therefore, it can analyze any given set of classes.

However, the analysis of an Android app is different from the analysis of an open library and it requires special consideration. Roughly speaking, we need to capture the “connections” among callback methods, or our inference might miss privacy leaks

```
public LocationLeak2 extends Activity implements LocationListener {
  private double latitude;
  protected void onResume() {
    double d = this.latitude; // TYPE ERROR!
    Log.d("Latitude", "Latitude: " + d); // sink
  }
  public void onLocationChanged(Location loc) {
    double lat = loc.getLatitue(); // loc is a source
    this.latitude = lat;
  }
}
```

Figure 6.14: LocationLeak2 refactored from DroidBench, highlights our inference's novel handling of callback methods.

through fields. Consider the LocationLeak2 example refactored from DroidBench in Figure 6.14. The secret lat of the current location, obtained in callback method onLocationChanged, flows through field latitude and reaches the public parameter of Log.d in another callback method, onResume. Local variables lat and d are secret and public, respectively. If the inference analyzed the app as a standard open library (e.g., as in [15]), it would infer this of onResume as public. This is because of (TREAD) constraint $\mathsf{this}_{\mathsf{onResume}} \rhd \mathsf{latitude} <: \mathsf{d}$ where $S(\mathsf{latitude}) = \{\mathsf{secret}, \mathsf{poly}\}$ and $S(\mathsf{d}) = \{\mathsf{public}\}$. Due to this constraint, $S(\mathsf{latitude})$ would be updated to $\{\mathsf{poly}\}$. Further, it would infer this of onLocationChanged as secret, because of (TWRITE) constraint $\mathsf{lat} <: \mathsf{this}_{\mathsf{onLocationChanged}} \rhd \mathsf{latitude}$ where $S(\mathsf{lat}) = \{\mathsf{secret}\}$. The inferred typing would type-check and the leak through field lattitude would be missed.

If the app were a standard open library, it would be composed with user code, which would instantiate the Activity and reveal the leak. Consider the following hypothetical user code

```
1  Activity a = new LocationLeak2();
2  a.onLocationChanged(loc);
3  a.onResume();
```

When composing this code with the inferred typing for LocationLeak2, there would be a type error because Statement 2 requires a to be secret (since this of onLocationChanged is mutable, there is equality constraint at 2: $\mathsf{a} = q^2 \rhd \mathsf{this}_{\mathsf{onLocationChanged}})$, while Statement 3 requires a to be public.

The Android app however, is not composed with user code. Instead, the Activity, as well as other component objects, are instantiated by the Android framework. Therefore, the inference needs to handle the implicit instantiation of app objects. The inference *creates equality constraints for* this *of all callback methods in the same class*, thus "connecting" this of callback methods. If the app type-checks, this means there is a solution for constraints

$$\begin{array}{rcl} q_{\mathsf{a}} & <: & q^{i1} \rhd q_{\mathsf{this}_{\mathsf{callback1}}} \\ q_{\mathsf{a}} & = & q^{i2} \rhd q_{\mathsf{this}_{\mathsf{callback2}}} \\ & \ldots & \end{array}$$

which correspond to the calls to callback methods in the Android framework.

In the LocationLeak2 example, the inference creates an equality constraint between the this parameters of onResume and onLocationChanged:

$$\mathsf{this}_{\mathsf{onResume}} = \mathsf{this}_{\mathsf{onLocationChanged}}$$

$\mathsf{this}_{\mathsf{onResume}}$ becomes secret. There is a type error at Statement 4, thus detecting the privacy leak.

6.2.7.3 Inter-Component Communication (ICC)

Android components (activity, service, broadcast receiver and content provider) interact through ICC objects — mainly *Intents.* Communication can happen across applications as well, to allow functionality reuse. There are two forms of Intent in Android:

- **Explicit Intents** have an explicit target component — the exact target class of the Intent is specified.
- **Implicit Intents** do not have a target component, but they include enough information for the system to implicitly determine the target component.

In the following code snippet

```
Intent i1 = new Intent();
Intent i2 = new Intent();
i1.setClassName("edu.rpi","edu.rpi.MyClass");
i2.setAction("edu.rpi.ACTION");
i2.addCategory("edu.rpi.CATEGORY");
```

i1 is an explicit Intent whose target component is "edu.rpi.MyClass", while i2 is an implicit Intent whose target component is determined by the information it includes.

Capturing data flow through Intents is important for detecting privacy leaks in Android. Consider the example refactored from a real malware app, Fakedaum[4] in Figure 6.15. The return value of SmsMessage.createFromPdu is a source and the parameter of HttpPost.setEntity is a sink. The broadcast receiver SmsReceiver intercepts the SMS messages, then puts the messages into an Intent and starts the background service TaskService with the Intent. Then TaskService sends the messages to the Internet without user consent. If the communication between the broadcast receiver SmsReceiver and the background service TaskService is not captured, there is no way to detect the privacy leak.

For explicit Intent whose target class is specified by a final or constant string, the inference connects the data carried by Intent using placeholders. Specifically, it replaces the Intent with a "typed" Intent at both the sender component and the receiver component. In addition, each putExtra and getExtra are treated as writing and reading a field in the "typed" Intent, respectively. Since the target class of Intent it in Figure 6.15 is specified by constant TaskService.class, the inference transforms the program into:

```
10 ...
11 TaskService_Intent it = new TaskService_Intent();
12 TaskService_Intent.data = sb.toString();
13 ...
18 String body = TaskService_Intent.data;
```

[4] `http://contagiominidump.blogspot.com/2013/11/fakedaum-vmvol-android-infostealer.html` (Retrieved on April 29, 2014)

```
public class SmsReceiver extends BroadcastReceiver {
  public void onReceive(Context c, Intent i) {
    Bundle bundle = intent.getExtras();
    Object[] pdusObj = (Object[]) bundle.get(”pdus”);
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < pdusObj.length; i++) {
      SmsMessage msg = SmsMessage.createFromPdu((byte[]) pdusObj[i]); //
          source
      String body = msg.getDisplayMessageBody();
      sb.append(body);
    }
    Intent it = new Intent(c, TaskService.class);
    it.putExtra(”data”, sb.toString());
    startService(i);
  }
}
public class TaskService extends Service {
  public void onStart(Intent it, int d) {
    String body = it.getSerializableExtra(”data”);
    List list = new LinkedList();
    list.add(body);
    HttpClient client = HttpClientManager.getHttpClient();
    HttpPost post = new HttpPost();
    post.setURI(URI.create(”http://103.30.7.178/getMotion.htm”));
    Entity e = new UrlEncodedFormEntity(list, ”UTF8”);
    post.setEntity(e); // sink
    client.execute(post);
  }
}
```

Figure 6.15: SMS message stealing in Fakedaum. The SMS message is intercepted in SmsReceiver and passed to TaskService via Intent. Finally, the message is sent out to the Internet using HTTP post method, resulting in a message leak.

As a result, the intercepted message is connected to the post data via placeholder data of TaskService_Intent. The leak is captured.

For explicit Intents whose target class is not specified by a constant string, a string analysis, which we leave for future work, is required to determine the target. The inference makes the worst-case assumption for such explicit Intents, as well as for implicit Intents carrying sensitive data, as their content can be intercepted by any, possibly malicious, component. This is achieved by annotating as public the Intent parameter of library methods that start new components, such as startActivity and startService. For example, if i2 refers to an implicit Intent carrying current location information, then there is a type error at statement startSerivce(i2) because startSerivce requires a public argument, but i2 is secret as it contains secret data.

CHAPTER 7
Empirical Results

The inference framework is built on top of the Checker Framework (CF) [18, 14]. The set-based solver generates the type constraints using CF for each statement, and solves the constraints using fixpoint iteration to produce the set-based solution. CF takes as input the Java source code, which unfortunately is not available for most Android apps, as they are usually delivered as an Android Package Files (APKs). Therefore, we extended the inference framework by building an Android constraint generation front-end, based on Soot [67] and Dexpler [68]. The architecture of the implementation of our inference framework is shown in Figure 7.1. The inference Framework is publicly available at `http://code.google.com/p/type-inference/`, including source.

In this section, we present the empirical results for ReIm, Ownership Types, Universe Types, SFlow/Integrity and SFlow/Confidentiality. All evaluations were performed on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (The maximal heap size is set to 2 GB.) The software environment consists of Oracle JDK 1.6, Checker Framework 1.3.0, Soot 2.5.0 nightly build on GNU/Linux 3.2.0.

7.1 ReIm

The ReIm instantiation of the inference framework, called ReImInfer is evaluated on 13 large Java benchmarks, including 4 whole-program applications and 9 Java libraries.

Whole programs:

Portions of this chapter previously appeared as: W. Huang *et al.*, "Inference and checking of object ownership," in *Proc. European Conf. Object-Oriented Programming*, Beijing, China, 2012, pp. 181–206.

Portions of this chapter previously appeared as: W. Huang *et al.*, "ReIm & ReImInfer: Checking and inference of reference immutability and method purity," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Tucson, AZ, 2012, pp. 879–896.

Portions of this chapter previously appeared as: W. Huang *et al.*, "Type-based taint analysis for Java web application," in *Int. Conf. Fundamental Approaches to Software Engineering*, Grenoble, France, 2014, pp. 140–154.

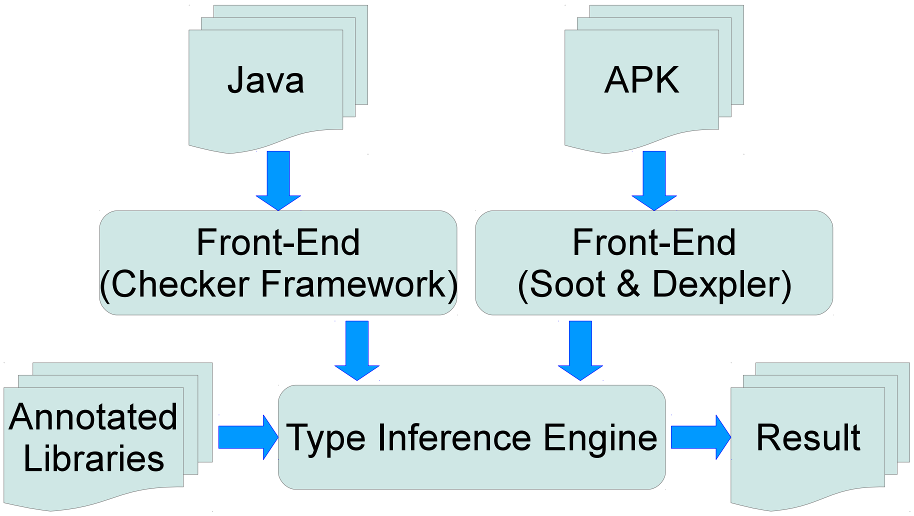


Figure 7.1: Architecture of the implementation of the inference framework.

- **Java Olden** (JOlden) is a benchmark suite of 10 small programs.
- **ejc-3.2.0** is the Java Compiler for the Eclipse IDE.
- **javad** is a Java class file disassembler.
- **SPECjbb 2005** is SPEC's benchmark for evaluating server side Java.

Libraries:

- **tinySQL-1.1** is a database engine.[5]
- **htmlparser-1.4** is a library for parsing HTML.
- **jdbm-1.0** is a lightweight transactional persistence engine.
- **jdbf-0.0.1** is an object-relational mapping system.
- **commons-pool-1.2** is a generic object-pooling library.
- **jtds-1.0** is a JDBC driver for Microsoft SQL Server and Sybase.
- **java.lang** is the package from JDK 1.6

[5] We added 392 empty methods in tinySQL in order to compile it with Java 1.6. The modified version is available online.

- **java.util** is the package from JDK 1.6.
- **xalan-2.7.1** is a library for transforming XML documents to HTML from the DaCapo 9.12 benchmark suite.

Benchmarks JOlden, tinySQL, htmlparser, and ejc are precisely the benchmarks used by Javarifier [6]. Javarifier's distribution includes regression tests, which greatly facilitates the comparison between Javarifier and ReImInfer. java.lang and java.util are included because they are representative libraries. The rest of the benchmarks come from our previous experimental work [69].

7.1.1 Experimental Setup

We treat the this parameters of java.lang.Object's hashCode, equal, and toString as readonly, even though these methods may mutate internal fields (these fields are used only for caching and can be excluded from the object state). This handling is consistent with the notion of *observational purity* discussed in [70] as well as other related analyses such as JPPA [29]; these methods are intended to be observationally pure. Our analysis does not detect bugs due to unintended mutation in these methods.

ReIm treats private fields f that are read and/or written through this in exactly one instance method m, as if they were local variables. Precisely, this means that for these fields we allow qualifier mutable, and treat field reads x = this.f and writes this.f = x as if they were assignments x = f and f = x. One such field and method are current and nextElement() in class Enumerate shown in Figure 7.2. We preserve the dependence between this and f, by using an additional constraint: $q_{\mathsf{this}} <: q_{\mathsf{f}}$. Thus, when f is mutated in m, f and this are inferred as mutable. When f is readonly in the scope of m, but depends on the context of the caller, f is polyread and this is polyread or mutable. If f is readonly, no constraints are imposed on this. As an example, field current and this of nextElement() in Figure 7.2 are both inferred polyread. The motivation behind this optimization is precisely the Enumeration class in Figure 7.2. The goal is to transfer the dependence from the element stored in the container, to the container itself, which is important for purity inference. If current were treated as a field, it would be polyread, and therefore, this of elements would be mutable,

```
public class Body {
  Body next;
  public final Enumeration elements() {
    class Enumerate implements Enumeration {
      private Body current;
      public Enumerate() { current = Body.this; }
      public Object nextElement() {
        Object retval = current;
        current = current.next;
        return retval;
      }
    }
    return new Enumerate();
  }
}
```

Figure 7.2: The elements() method in JOlden/BH

which entails that every container that creates an enumeration is mutable, even if its elements were not mutated. If current was excluded from abstract state, then this of nextElement would have been readonly and mutation from elements would not have been transferred to the container. Our optimization allows this of nextElement and elements to be polyread, which is important for purity inference, as we discuss shortly. The optimization affected 8 nextElement and elements methods and 12 other methods that call nextElement and elements throughout all of our benchmarks.

Recall that reference immutability inference is modular. Thus, it is able to analyze any given set of classes L. If there are unknown callees in L, the analysis assumes default typing mutable, mutable → polyread. The mutable parameters assume worst-case behavior of the unknown callee — the unknown callee mutates its arguments. Clearly, readonly is the most general return type. However, this will require that every use of the return in the client is readonly, and many clients violate this restriction. mutable, mutable → polyread is safe because we can always assign the polyread return value to a readonly variable. And it also imposes a constraint on the callee: e.g., suppose the code for X id(X p) { return p; } was unavailable and we assumed typing mutable → polyread for id. When it becomes available, p will be polyread.

User code U, which uses previously analyzed library L, is analyzed separately using the result of the analysis of L. In our case, when analyzing user code U, we use the annotated JDK available with Javarifier from the CF; the similarities between Javari and ReIm justify this use. Correctness of the composition is ensured by the check that the method overriding constraints (see Section 2.2.3) hold: for every m' in U that overrides an m from L, $\mathit{typeof}(\mathsf{m}') <: \mathit{typeof}(\mathsf{m})$, i.e.

$$(q_{\mathsf{this}_{\mathsf{m}'}}, q_{p_{\mathsf{m}'}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}'}}) \;\; <: \;\; (q_{\mathsf{this}_{\mathsf{m}}}, q_{p_{\mathsf{m}}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}}})$$

must hold, which means that $q_{\mathsf{this}_{\mathsf{m}}} <: q_{\mathsf{this}_{\mathsf{m}'}}, q_{p_{\mathsf{m}}} <: q_{p_{\mathsf{m}'}}$, and $q_{\mathsf{ret}_{\mathsf{m}'}} <: q_{\mathsf{ret}_{\mathsf{m}}}$ must hold. For example, suppose that L contains code $\mathsf{x.m()}$ where $\mathsf{this}_{\mathsf{m}}$, is inferred as readonly. The typing is correct even in the presence of callbacks. If $\mathsf{x.m()}$ results in a callback to m' in U (m' overrides m), constraint $\mathit{typeof}(\mathsf{m}') <: \mathit{typeof}(m)$ which entails $\mathsf{this}_{\mathsf{m}} <: \mathsf{this}_{\mathsf{m}'}$, ensures that $\mathsf{this}_{\mathsf{m}'}$ is readonly as well.

Of course, it is possible that U violates the subtyping expected by L. Interestingly however, in our experiments the only violations were on special-cased methods of Object: equals, hashCode and toString. Furthermore, the vast majority of violations occurred in the java.util library. As with other analyses (JPPA), we report these violations as warnings.

7.1.2 Inference Result

Below, we present our results on inference of reference immutability. Table 7.1 presents the result of running our inference on all benchmarks.

Inference Output In all benchmarks, about 41% to 69% of references are reported as readonly, less than 16% are reported as polyread and 24% to 50% are reported as mutable.

Timing Results Figure 7.3 compares the running times of ReImInfer and Javarifier on the first 5 benchmarks in Table 7.1. ReImInfer and Javarifier analyze exactly the same set of classes (given at the command-line), and use stubs for the JDK. That is, both ReImInfer and Javarifier generate and solve constraints for the exact same set

Table 7.1: Inference results for reference immutability. #Line shows the number of lines of the benchmarks, including blank lines and comments. Annotatable References include all references, including fields, local variables, return values, formal parameters, and implicit parameters this. It does not include variables of primitive type. #Ref is the total number of annotatable references, #Readonly, #Polyread, and #Mutable are the number of references inferred as readonly, polyread, and mutable, respectively. We also include the running time for the benchmarks. The last column Time shows the total running time in seconds, including reference immutability inference and type-checking.

		Annotatable References				
Benchmark	**#Line**	**#Ref**	**#Readonly**	**#Polyread**	**#Mutable**	**Time**
JOlden	6223	949	453 (48%)	149 (16%)	347 (37%)	5.7
tinySQL	31980	4247	2644 (62%)	418 (10%)	1185 (28%)	15.1
htmlparser	62627	4853	2711 (56%)	421 (9%)	1721 (35%)	16.9
ejc	110822	15434	6161 (40%)	1803 (12%)	7470 (48%)	66.2
xalan	348229	41186	25181 (61%)	3254 (8%)	12751 (31%)	81.1
javad	4207	363	249 (69%)	19 (5%)	95 (26%)	3.2
SPECjbb	28333	1537	830 (54%)	246 (16%)	461 (30%)	9.3
commons-pool	4755	602	266 (44%)	37 (6%)	299 (50%)	3.8
jdbm	11610	1161	470 (40%)	161 (14%)	530 (46%)	5.9
jdbf	15961	2510	1669 (66%)	240 (10%)	601 (24%)	9.6
jtds	38064	5048	2805 (56%)	299 (6%)	1944 (39%)	17.2
java.lang	43282	2970	2028 (68%)	187 (6%)	755 (25%)	12.1
java.util	59960	6920	2852 (41%)	1005 (15%)	3063 (44%)	24.5

of classes, and neither analyzes the JDK. The timings are the medians of three runs.

ReImInfer scales better than Javarifier. ReImInfer appears to scale approximately linearly. As the applications grow larger, the difference between ReImInfer and Javarifier becomes more significant. These results are consistent with the results reported by Quinonez et al. [16] where Javarifier posts significant nonlinear growth in running time, when program size goes from 62kLOC to 110kLOC.

7.1.3 Correctness and Precision Evaluation

To evaluate the correctness and precision of our analysis, we compared our result with Javarifier on the first four benchmarks from Table 7.1. We do not compare the numbers directly because we use a different notion of annotatable reference from Javarifier (e.g., Javarifier counts List<Date> twice while we only count it once). In our comparison, we examine only fields, return values, formal parameters, and this parameters; we call these references *identifiable references.* We exclude local

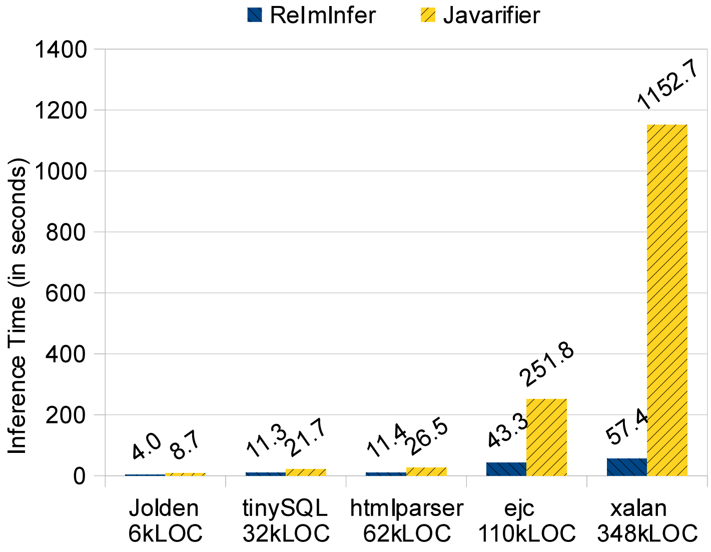


Figure 7.3: Runtime performance comparison. Note that the running time for type-checking is excluded for both ReImInfer and Javarifier.

variables because Javarifier does not give identifiable names for local variables (it only shows local 0, local 1, and so on). In addition, polyread fields in Javarifier are called this-mutable. In the comparison, we view all such fields as polyread.

JOlden We examined all programs in the Java Olden (JOlden) benchmark suite. We found 34 differences between our result and Javarifier's, out of 758 identifiable references. We exclude the following difference from the count: the this parameters of constructors are reported as readonly by Javarifier, while they are reported as mutable if this is mutated, by ReImInfer. Differences due to the annotated JDK are also excluded because Javarifier treated variables from library methods as mutable even though we have specified the annotated JDK. 8 out of the 34 differences are the nextElement() method that implements the Enumeration interface (Figure 7.2). Javarifier infers the return value as readonly. This is correct with respect to the semantics of Javari and Javarifier, which separates a structure from the elements stored in it; thus, a mutation on an element, should not necessarily affect the data structure itself.

The semantics of ReIm and ReImInfer demands that the return of nextElement should be polyread, because there are cases when the retrieved element is mutated. ReImInfer reports that nextElement()'s return is polyread. Also Javarifier infers the this parameter of nextElement() as mutable while ReImInfer reports that it is polyread. This is possible because ReImInfer treats field current in Figure 7.2 as a local variable as discussed earlier. There are 4 nextElement() methods in the JOlden benchmark suite, causing 8 differences in total.

These 8 differences directly or indirectly lead to the remaining 26 differences. First, these 8 differences directly lead to 8 differences in the current field and the elements() method in the Enumerate class, which is shown in Figure 7.2. Our analysis infers retval as polyread because the return value of nextElement() is polyread as discussed earlier. This causes field current to be inferred as polyread in statement Object retval=current since current is a field but treated as a local variable. As a result, the this parameter of elements() becomes polyread due to the assignment current=Body.this. Because Javarifier infers the return value of nextElement() as readonly, it reports both the current field and the this of elements() are readonly, which leads to 8 differences in total. The treatment of Javarifier reflects the expected semantics of Javari — the container that calls elements should not be affected by the data stored in it. ReIm and ReImInfer's semantics demands that a mutation on the element is propagated to the container.

Second, these 8 differences on current and elements() propagate to the other 18 differences. The following code shows an example:

```
Body bodyTab = ...;
for(Enumeration e = bodyTab.elements(); e.hasMoreElements();) {
  Body b = (Body)e.nextElement();
  ...
  b.setProcNext(prev);
}
```

Here b is mutable since the this parameter of setProcNext(Body) is mutable. Because bodyTab is indirectly assigned to b through the Enumeration instance referred by e, bodyTab should be mutable as well. Javarifier reports bodyTab is readonly because the this parameter of elements() is inferred as readonly. ReImInfer reports bodyTab as mutable. This is important for purity — e.g., if bodyTab is a parameter, its mutability entails that the enclosing method is impure.

Other benchmarks For the remaining three benchmarks, tinySQL, htmlparser and ejc, we examined 4 randomly selected classes from each (a total of 12 classes). We found 2 differences out of 868 identifiable references. The 2 differences are caused by the fact that Javarifier infers a parameter of String type as polyread, which causes an actual argument to become polyread or mutable; ReImInfer infers this parameter as readonly.

Overall, the differences are very minor. Most are attributable to the different semantics of ReIm and Javari, and the few others are due to an apparent bug in a corner case of Javarifier's handling of the annotated JDK.

7.1.4 Purity Inference

This section presents our results on purity inference. We treat methods equals, hashCode, toString in java.lang.Object, as well as java.util.Comparable.compareTo, as observationally pure. This is analogous to previous work [29].

Our purity inference is modular. Reference immutability assumptions for unknown callees are exactly as before. We ran ReImInfer on java.lang and java.util packages, and we assumed that other library methods have not mutated static fields. JPPA, a Java Pointer and Purity Analysis tool by Sălcianu and Rinard [29], makes the same assumption for unknown library methods, and our decision to use $q_{\mathsf{m}} =$ readonly as default, is motivated by this, in order to facilitate comparison with JPPA.

When composing previously analyzed libraries L with user code U for purity inference, we need one additional check: for every m' in U that overrides m in L, we must have $q_{\mathsf{m}} <: q_{\mathsf{m}'}$. In particular, if q_{m} is inferred as readonly, then $q_{\mathsf{m}'}$ must be readonly as well. As with reference immutability, it is possible that user code violates this constraint. In the first 11 benchmarks in Table 7.1, we found 205 out of 22,720 user methods that violate the inferred *statictypeof* on java.lang and java.util packages, and the vast majority of the violations are on the special-cased methods, equals, hashCode, and toString. These violations are reported as warnings.

The results of purity inference by ReImInfer are shown in Table 7.1, column **#Pure**. To evaluate analysis precision, we compared with JPPA by Sălcianu and Rinard [29] and JPure by Pearce [71]. We ran JPPA and JPure on the JOlden

Table 7.2: Pure methods in Java Olden benchmarks.

Program	#Meth	JPPA	JPure	ReImInfer
BH	69	20 (29%)	N/A	33 (48%)
BiSort	13	4 (31%)	3 (23%)	5 (38%)
Em3d	19	4 (21%)	1 (5%)	8 (42%)
Health	26	6 (23%)	2 (8%)	11 (42%)
MST	33	15 (45%)	12 (36%)	16 (48%)
Perimeter	42	27 (64%)	31 (74%)	38 (90%)
Power	29	4 (14%)	2 (7%)	10 (34%)
TSP	14	4 (29%)	0 (0%)	1 (7%)
TreeAdd	10	1 (10%)	1 (10%)	6 (60%)
Voronoi	71	40 (56%)	30 (42%)	47 (66%)

benchmark suite and directly compared its output with ours. Table 7.2 presents the comparison results.

To summarize our results, ReImInfer scales well to large programs and shows good precision compared to JPPA and JPure. Furthermore, ReImInfer, which is based on the stable and well-maintained CF, appears to be more robust than JPPA and JPure, both of which are based on custom compilers. These results suggest that ReImInfer can be useful in practice.

7.1.4.1 Comparison with JPPA

JOlden There are 59 differences out of 326 user methods between ReImInfer's result and JPPA's. Of these differences, (a) 4 are due to differences in definitions/assumptions, (b) 51 are due to limitations/bugs in JPPA and (c) 4 are due to limitations in ReImInfer.

4 differences are due to JPPA's assumption about unknown library methods. For example, JPPA reports as pure the method median in Jolden/TSP, which invokes new java.lang.Random(). The constructor Random should not be pure because it mutates a static field seedUniquifier. ReImInfer precomputes static immutability types q_{m} on the JDK library and thus reports method median as impure.

51 differences are due to limitations/bugs of JPPA. 38 differences are the constructors, which ReImInfer reports as pure but JPPA does not. According to [29], JPPA follows the JML convention and constructors that mutate only fields of the this object are pure. Thus, JPPA should have inferred them as pure. ReImInfer follows

the same definition and reports these constructors as pure. There are 3 differences on methods that are inferred as pure by ReImInfer but impure by JPPA. These 3 methods return newly-constructed objects, which are mutated later. According to the definition in [29], JPPA should have inferred them as pure. There is 1 difference on method loadTree in Jolden/BH. It is likely a bug in JPPA because the this parameter is passed to another object's field which is mutated later, but JPPA reports loadTree as pure. ReImInfer detects the this parameter is mutated and reports the method as impure. There are 9 methods reported as pure by ReImInfer but not covered by JPPA. This is because JPPA is a whole-program analysis and these methods are not reachable, resulting in 9 differences in the comparison.

The remaining 4 differences are the nextElement method discussed in Section 7.1.1. Because ReImInfer considers the current field as a local variable, it infers these 4 methods as pure while JPPA considers they are impure.

Other benchmarks We attempted to run JPPA and compare on benchmarks tinySQL, htmlparser, and ejc as we did with Javarifier. tinySQL is a library and there is no main method. htmlparser, which is a library as well, comes with a main, which exercises a portion of its functionality; JPPA threw an exception on htmlparser which we were unable to correct. JPPA completed on ejc. Due to the fact that it is a whole-program analysis, it analyzed 3790 reachable user methods; ReImInfer covered all 4734 user methods.

We examined 4 randomly selected classes from ejc and found 17 differences out of 163 methods in total. 9 methods are not reachable according to JPPA. Of the remaining 8 differences, (a) 2 are due to limitations/bugs in JPPA and (b) 6 are due to limitations/bugs in ReImInfer. 1 constructor that should have been pure according to the JML convention was reported as impure by JPPA. In addition, 1 method which we believe is pure because it does not mutate any prestate, was reported as impure by JPPA. The remaining 6 methods are reported as pure by JPPA but impure by ReImInfer; this is imprecision in ReImInfer. These methods are inferred as impure by ReImInfer because they are overridden by impure methods. This is an insurmountable imprecision for ReImInfer.

7.1.4.2 Comparison with JPure

JOlden There are 60 differences out of 257 user methods between ReImInfer's result and JPure's, excluding the BH program (JPure could not compile BH). Of these, (a) 29 differences are caused by different definitions/assumptions, (b) 2 are caused by limitations/bugs in ReImInfer, and (c) 29 differences are caused by limitations/bugs in JPure.

29 differences are caused by different definitions of pure constructors. We follow the JML convention that a constructor is pure if it only mutates its own fields. JPure has different definition of a pure constructor and that leads to these differences. 2 differences are the nextElement method where ReImInfer considers the current field as a local variable as discussed above. There are 8 differences in toString methods, which are inferred as impure by JPure. Our examination shows that those methods are pure; it appears that they should be pure, but are inferred as impure due to imprecision in JPure, according to [71]. 16 differences are caused by methods that return fresh local references. JPure should have been able to identify them as @Fresh, but it did not. The remaining 5 differences are due to the static methods in java.lang.Math. JPure infers all methods that invoke the static methods in java.lang.Math as impure, while ReImInfer identifies that these methods satisfy q_{m} is readonly by using the inference result from the java.lang package.

Other benchmarks We attempted to run JPure on the libraries from JDK 1.6, but that caused a problem with the underlying compiler in JPure. We attempted to run JPure on tinySQL, htmlparser and ejc. In all three cases, the tool issued an error. We were unable to perform direct comparison on larger benchmarks.

7.2 Universe Types and Ownership Types

7.2.1 Experimental Setup

We evaluated our implementation using eight Java programs of up to 110kLOC (see Table 7.3). The analysis processes only application code. References from libraries receive $\{\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ for Ownership Types and $\{\mathsf{peer}\}$ for Universe Types in the initial set-based solution. The analysis is modular, in the sense that it can

Table 7.3: The benchmarks used by Ownership Types and Universe Types.

Benchmark	#Lines	#Meths	Description
JOlden	6223	326	Benchmark suite of 10 small programs
tinySQL	31980	1597	Database engine
htmlparser	62627	1698	HTML parser
ejc	110822	4734	Compiler of the Eclipse IDE
javad	4207	140	Java class file disassembler
SPECjbb	12076	529	SPEC's benchmark for evaluating server side Java
jdepend	4351	328	Java package dependency analyzer
classycle	8972	440	Java class and package dependency analyzer

Table 7.4: The inference results for Universe Types. Column #Ref gives the total number of references excluding implicit parameters this. Column #Pure gives the number of pure methods inferred automatically based on reference immutability [15]. Columns #any, #rep, and #peer give the number of references inferred as any, rep, and peer, respectively. No user annotations are needed for the inference of Universe Types; therefore, there are only zeros in the #Man column. Last column Time shows the total running time in seconds including parsing the source code, type inference, and type-checking.

Benchmark	#Pure	#Ref	#any	#rep	#peer	#Man	Time
JOlden	175	685	227 (33%)	71 (10%)	387 (56%)	0	11.3
tinySQL	965	2711	630 (23%)	104 (4%)	1977 (73%)	0	18.2
htmlparser	642	3269	426 (13%)	153 (5%)	2690 (82%)	0	22.9
ejc	1701	10957	1897 (17%)	122 (1%)	8938 (82%)	0	119.7
javad	60	249	31 (12%)	11 (4%)	207 (83%)	0	4.1
SPECjbb	195	1066	295 (28%)	74 (7%)	697 (65%)	0	13.6
jdepend	102	542	95 (18%)	14 (3%)	433 (80%)	0	7.2
classycle	260	946	87 (9%)	11 (1%)	848 (90%)	0	9.9

analyze whatever code is available, including libraries with no main method.

7.2.2 Inference Result of Universe Types

Inference of Universe Types requires information about method side effects. We used our purity inference tool described in the previous section. The purity inference relies on ReIm. The maximal typing always type-checks for Universe Types, and the set-based solver infers the unique maximal typing.

Table 7.4 shows the inference results for Universe Types. Across all benchmarks, 9%–33% of all variables are inferred as any, the best qualifier. 1% to 10% of all variables are inferred as rep. A relatively large percentage (57%–92%) of the variables

Table 7.5: The inference results for Ownership Types. Column #Ref again gives the total number of references excluding the implicit parameters this. Columns #⟨rep|_⟩, #⟨own|_⟩, #⟨p|_⟩, and #⟨norep|_⟩ give the numbers of variables whose owners are inferred as rep, own, p, and norep, respectively. The boldfaced number in parentheses in column #⟨rep|_⟩ is an upper bound on rep typings; it is discussed in the text. #Man shows the total number of manual annotations and, in parentheses, the number of annotations per 1kLOC. Time shows the running time in seconds.

Benchmark	#Ref	#⟨rep|_⟩	#⟨own|_⟩	#⟨p|_⟩	#⟨norep|_⟩	#Man	Time
JOlden	685	67 (10%/**10%**)	497 (73%)	24 (4%)	97 (14%)	13 (2)	10.3
tinySQL	2711	224 (8%/**11%**)	530 (20%)	5 (0%)	1952 (72%)	215 (7)	18.4
htmlparser	3269	330 (10%/**11%**)	629 (19%)	36 (1%)	2274 (70%)	200 (3)	33.6
ejc	10957	467 (4%/**4%**)	1768 (16%)	50 (0%)	8672 (79%)	592 (5)	122.4
javad	249	44 (18%/**19%**)	27 (11%)	74 (30%)	104 (42%)	46 (10)	5.5
SPECjbb	1066	166 (16%/**16%**)	141 (13%)	71 (7%)	688 (65%)	73 (6)	17.1
jdepend	542	130 (24%/**25%**)	156 (29%)	128 (24%)	128 (24%)	26 (6)	13.7
classycle	946	153 (16%/**20%**)	173 (18%)	28 (3%)	592 (63%)	90 (10)	11.7

are inferred as peer, resulting in a flat ownership structure. This is consistent with the results by Dietl et al. [17]. There are several possible reasons that lead to flat ownership structures. One is due to utility methods whose formal parameters are passed to impure methods. This forces the formal parameters to be peer. Another reason is that the inference uses the default peer annotation for libraries.

7.2.3 Inference Result of Ownership Types

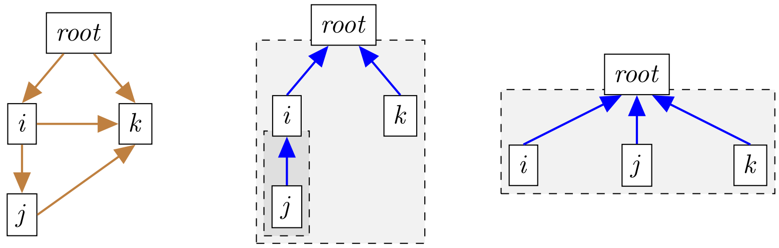
In OT, we add an additional modifier norep, which refers to *root*, as described in detail in [72]. We use norep as the default type for String and boxed primitives such as Boolean, Integer, etc.

Table 7.5 shows the inference results for OT. Note that there are many ⟨norep|_⟩ variables; the majority of these are strings and boxed primitives, e.g. 521 out of 688 ⟨norep|norep⟩ variables in SPECjbb are strings and boxed primitives whose default type is norep.

Compared to UT, a relatively large percentage (4%–24%) of variables are inferred as $\langle \textsf{rep}|\_\rangle$ in OT. Note however, that this *does not imply a deeper ownership tree compared to UT*. In UT, many of the any variables can refer to a rep object (as UT distinguishes readonly access); in contrast, in OT only a rep variable can refer to a rep object.

Due to the fact that the maximal typing does not always type-check for OT, the inference requires manual annotations. Column #Man gives the total numbers of manual annotations that were added and, in parentheses, the number of annotations per 1kLOC. The annotation burden is low — on average, 6 annotations per 1kLOC. Although the set-based solver cannot produce a maximal typing automatically, it is quite valuable, because it reduces the burden of annotations on programmers. The set-based solver prints all conflicts and lets the programmer choose an annotation that resolves the conflict in such a way that it reflects their intent. This process continues until all conflicts are resolved. By doing so, we annotated JOlden (6223 LOC) in approximately 10 minutes and SPECjbb in approximately 2 hours. The annotations reflect the intent of the first author, but not necessary the intent of the programmers of these benchmarks. Finally, the last column Time shows the time in seconds to do type inference and type checking *after* the manual annotations. It is approximately equal to the initial run that outputs all conflicts and does not include the time to annotate the benchmark.

The boldfaced percentage shown in parentheses in column $\#\langle \textsf{rep}|\_\rangle$, is the percentage of all references that contain a $\langle \textsf{rep}|\textsf{rep}\rangle$, $\langle \textsf{rep}|\textsf{own}\rangle$ or $\langle \textsf{rep}|\textsf{p}\rangle$ in their set-based solution. This is an upper bound on the possible rep typings: even an ownership type system with many ownership parameters will be unable to type a larger percentage of variables as rep. The fact that the percentage of $\#\langle \textsf{rep}|\_\rangle$'s in our typing is close to this bound, has two implications: (1) our typing is precise, and (2) one ownership parameter may be sufficient in practice (again, if the goal is to maximize the number of rep typings).



(a) Object graph (b) OT tree (c) UT tree

Figure 7.4: Write access to enclosing context results in flatter structure for UT as compared to OT. The bold edge from j to k highlights the write access.

7.2.4 Comparing Universe Types vs. Ownership Types

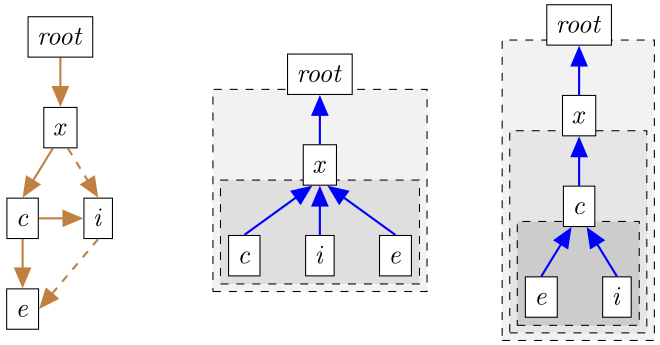
In this section, we compare Universe Types, which enforce the owner-as-modifier encapsulation discipline, to Ownership Types, which enforce the owner-as-dominator encapsulation discipline, using examples we observed in the benchmarks.

In some cases, Universe Types inferred flatter structures than Ownership Types. This happens when an object j modifies an object k in an enclosing context. For example, consider Figure 7.4. If object j *modifies* k, j and k must be peers in UT, which will force the flat ownership tree in Figure 7.4(c). In contrast, OT reflects dominance and produces the deeper ownership tree shown in Figure 7.4(b).

In other cases, Ownership Types inferred flatter structures than Universe Types. OT disallows exposure of internal objects outside of the boundary of the owner. UT is more permissive, in the sense that it allows readonly exposure. Consider Figure 7.5. which represents a container c, its internal representation e and an iterator i over e. The OT tree is flatter because the iterator i creates a path to e which does not go through c. Therefore, c, e, and i must have x as their owner. In contrast, UT allows the exposure of i to x because this exposure is readonly. Therefore, c remains the owner of both e and i.

Table 7.6 compares OT and UT on the benchmarks. We consider only allocation sites, excluding strings and boxed primitives. Allocation sites provide the best approximation of ownership structure.

On average 25% of the OT $\langle \mathsf{rep}|\_\rangle$ sites are typed rep in UT as well. On the



(a) Object graph (b) OT tree (c) UT tree

Figure 7.5: Readonly sharing of internal representation results in flatter structure for OT as compared to UT. The dotted edge from i to e highlights the readonly access.

Table 7.6: Ownership Types vs. Universe Types on allocation sites. The four columns give the number of OT/UT pairings and, in parenthesis, the corresponding percentages. For example, column ⟨rep|_⟩/peer shows the number of allocation sites that were inferred as rep in Ownership Types and peer in Universe Types.

Benchmark OT:	⟨rep|_⟩	⟨rep|_⟩	not ⟨rep|_⟩	not ⟨rep|_⟩
UT:	rep	peer	rep	not rep
JOlden	26 (22%)	8 (7%)	19 (16%)	66 (55%)
tinySQL	32 (6%)	123 (24%)	13 (2%)	355 (68%)
htmlparser	27 (2%)	234 (20%)	16 (1%)	926 (77%)
ejc	44 (2%)	336 (12%)	81 (3%)	2321 (83%)
javad	6 (10%)	38 (66%)	0 (0%)	14 (24%)
SPECjbb	75 (26%)	84 (29%)	25 (9%)	110 (37%)
jdepend	13 (7%)	71 (41%)	1 (1%)	90 (51%)
classycle	1 (0%)	109 (45%)	5 (2%)	128 (53%)

other hand, on average 64% of the UT rep sites are typed ⟨rep|_⟩ in OT as well. The discrepancy shows that it may be more common to have write access to enclosing context (which lowers rep to peer in UT), than it is to have readonly sharing of internal structure (which allows an object to stay rep in UT while it is not rep in OT). On average 40% of all allocation sites are inferred as rep in OT, and 14% are inferred as rep in UT, which suggests that write access to enclosing context is more common than readonly sharing of internal structure. The results suggest that in general, UT and OT capture distinct ownership structure. Note that as expected,

Table 7.7: Information about benchmarks and running time of SFlow-Infer. The file and line counts include Java files precompiled from JSP files. The time is for running configuration [*Parameter manipulation, SQL injection*]. The time for running other configurations is practically the same.

Benchmark	Version	#File	#Line	Time (s)
blojsom	1.9.6	61	12830	15.1
blueblog	1.0	31	4139	7.5
friki	2.1.1	21	1843	4.5
gestcv	1.0	119	7422	10.1
jboard	0.3	89	17405	22.2
jspwiki	2.4	364	83329	126.9
jugjobs	alpha	25	4044	18.7
pebble	1.6beta1	234	42542	50.3
personalblog	1.2.6	68	9943	17.6
photov	2.1	129	126886	640.2
roller	0.9.9	276	81171	213.4
snipsnap	1.0beta	488	73295	87.3
webgoat	0.9	35	8474	9.6

there is a significantly larger percentage of rep allocation sites in UT compared to rep variables.

7.3 SFlow/Integrity

The SFlow/Integrity instantiation of the inference framework, called SFlowInfer is evaluated on 13 relatively large Java web applications, used in previous work [43, 44, 45]. The benchmarks are listed in Table 7.7.

7.3.1 Experimental Setup

We use the sources and sinks described in detail in Livshits and Lam [43, 73]. In addition, we use 59 sources and sinks in API methods of Struts, Spring, and Hibernate, discovered as described in Section 6.1.7. There are 3 categories of sources [43]: *Parameter manipulation*, *Header manipulation*, and *Cookie poisoning*. There are 4 categories of sinks [43]: *SQL injection*, *HTTP splitting*, *Cross-site scripting (XSS)*, and *Path traversal*. These sources and sinks are added to the annotated JDK, Struts, Spring, and Hibernate, which is easily done with the CF. Once these annotated libraries are created, individual web applications are analyzed without any input from the user.

7.3.2 Inference Result

We run the benchmarks with all 12 configurations. Table 7.7 presents the sizes of the benchmarks as well as the running times of SFlowInfer in seconds. The running times attest to efficiency — for all but 1 benchmark, the analysis completes in less than 4 minutes; we believe that these running times can be improved.

We examined the type errors reported by SFlowInfer, and classified them as Type-1 (**T1**), Type-2 (**T2**), or False-positive (**FP**). Type-1 errors reflect direct flow from a source to a sink. The following code, adapted from webgoat, is a Type-1 error for configuration [*Parameter,SQL*]:

```
String u = request.getParameter(“user”); //source
String s = “SELECT ∗ FROM users WHERE name = ” + u;
stat.executeQuery(s); //sink, type error!
```

Another example of a Type-1 error, adapted from benchmark blueblog, is shown below. This is a type error for configuration [*Parameter,Path*]. This example illustrates a complex flow that goes through heap objects and method calls. It attests to the power of our analysis.

```
class BBServlet {
   ...
   String title = request.getParameter(”title”); //source
   String content = ...
   BlogData bd = new BlogData(title, content);
   currentCategory.addNewBlog(bd);
   ...
}
class FSCategory extends Category {
  ...
  Blog addNewBlog(BlogData bd) {
    ...
    return FSBlog.createNewBlog(...,bd,...);
  }
}
class FSBlog extends Blog {
  static FSBlog createNewBlog(...,BlogData blogData,...) {
    String filename = blogData.getSuggestedId(); //type error!
    File file = new File(filename+fileEndings); //sink
    ...
  }
}
```

```
23 class BlogData {
24   String title;
25   String suggestedId;
26   BlogData(String title, String content) {
27     this.title = title;
28     this.suggestedId = constructSuggestedId(title);
29     ...
30   }
31 }
```

Observe the complex flow from the source at line 3 to the sink at line 19. The servlet creates a new BlogData object, and passes the tainted title to it. Fields title and suggestedId of the BlogData object store tainted values. The BlogData object is then passed as argument to addNewBlog in FSCategory (line 6) and then to createNewBlog in FSBlog (line 13). createNewBlog reads the suggestedId field of the BlogData object and sends it to the sink. SFlowInfer reports a type error at line 18.

Type-2 errors reflect key-value dependences. The following code, adapted from personalblog, is a Type-2 error for configuration [*Parameter,XSS*]:

```
HashMap map = ...; PrintWriter out = ...;
String id = request.getParameter(''id''); //source
User user = (User) map.get(id);
out.print(user.getName()); //sink, type error!
```

The tainted id is used as a key to retrieve the user from the map, then user.getName() is sent to a safe sink (the parameter of PrintWriter.print). This is a dangerous flow according to the semantics of noninterference, because the tainted value of the key affects the value of the safe sink.

We classified as **FP** all errors that we could not easily identify as Type-1 or Type-2. The results over the 12 configurations are presented in Table 7.8, Table 7.9 and Table 7.10.

Table 7.8: Inference results for [*Parameter*, *SQL*], [*Parameter*, *XSS*], [*Parameter*, *HTTP*] and [*Parameter*, *Path*]. The multicolumns show numbers of Type-1 (T1), Type-2 (T2), and False-positive (FP) type errors for the four configurations; note that a large number of benchmarks have 0 type errors, i.e., they are proven safe.

	[*Parameter*,*SQL*]			[*Parameter*,*XSS*]			[*Parameter*,*HTTP*]			[*Parameter*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	1	0	0 (0%)	10	1	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	3	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	1	0	9 (90%)	8	1	0 (0%)
gestcv	1	0	0 (0%)	0	8	2 (20%)	0	0	0 (0%)	1	0	0 (0%)
jboard	3	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	25 (100%)	73	12	20 (19%)	23	0	16 (34%)	72	0	23 (24%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	2	0	0 (0%)	4	0	3 (37%)	43	3	0 (0%)
personalblog	6	0	0 (0%)	3	21	2 (8%)	0	0	0 (0%)	0	0	0 (0%)
photov	46	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	21	2	0 (0%)	1	2	1 (25%)	0	5	19 (79%)
snipsnap	0	0	3 (100%)	1	0	0 (0%)	6	0	0 (0%)	8	26	13 (28%)
webgoat	10	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	1	0	4 (80%)
Average			(**15%**)			(**4%**)			(**14%**)			(**16%**)

Table 7.9: Inference results for [*Header*, *SQL*], [*Header*, *XSS*], [*Header*, *HTTP*] and [*Header*, *Path*]. The multicolumns show numbers of Type-1 (T1), Type-2 (T2), and False-positive (FP) type errors for the four configurations. Again a large number of benchmarks have 0 type errors, i.e., they are proven safe. Due to time constraints, we did not examine the type errors for jspwiki; instead, we conservatively classified them as False-positive. Therefore, the actual False-positive rate is lower than the one reported.

	[*Header*,*SQL*]			[*Header*,*XSS*]			[*Header*,*HTTP*]			[*Header*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	3 (100%)
gestcv	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jboard	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	53? (100%)	0	0	113? (100%)	0	0	50? (100%)	0	0	154? (100%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
personalblog	1	0	0 (0%)	0	16	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
photov	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	1	0	0 (0%)	1	0	0 (0%)	0	0	0 (0%)
snipsnap	0	0	0 (0%)	7	0	0 (0%)	2	0	0 (0%)	0	25	54 (68%)
webgoat	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
Average			(**8%**)			(**8%**)			(**8%**)			(**21%**)

Table 7.10: Inference results for [*Cookie*, *SQL*], [*Cookie*, *XSS*], [*Cookie*, *HTTP*] and [*Cookie*, *Path*]. The multicolumns show numbers of Type-1 (T1), Type-2 (T2), and False-positive (FP) type errors for the four configurations. Again, we conservatively classified all errors in jspwiki as False-positive and the actual False-positive rate is lower than the one reported.

	[*Cookie*,*SQL*]			[*Cookie*,*XSS*]			[*Cookie*,*HTTP*]			[*Cookie*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
gestcv	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jboard	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	53? (100%)	0	0	172? (100%)	0	0	50? (100%)	0	0	155? (100%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
personalblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
photov	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	1 (100%)
snipsnap	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	19	8 (30%)
webgoat	1	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
Average			(**8%**)			(**8%**)			(**8%**)			(**18%**)

Table 7.11: Summary of comparison on DroidBench [47] with other taint analysis tools ($\surd$ = correct warning, $\times$ = false warning, $\bigcirc$ = missed flow).

Tool Name	**AppScan Source**	**Fortify SCA**	**FlowDroid**	**DroidInfer**
Sum, Precision and Recall–excluding implicit flows				
$\surd$, higher is better	14	17	26	26
$\times$, lower is better	5	4	4	8
$\bigcirc$, lower is better	14	11	2	2
Precision $p = \surd/(\surd + \times)$	74%	81%	86%	76%
Recall $r = \surd/(\surd + \bigcirc)$	50%	61%	93%	93%
F-measure $2pr/(p + r)$	0.60	0.70	0.89	0.84

7.4 SFlow/Confidentiality

The SFlow/Confidentiality instantiation of the inference framework, called DroidInfer, is evaluated on three sets of Android apps: 1) DroidBench [47], 2) apps from the official Google Play Store [74], and 3) malware from the contagio website [75].

7.4.1 DroidBench

We run DroidInfer on DroidBench, which is a suite of 39 Android apps designed by Fritz et al. [47] for the purpose of evaluating taint analysis for Android. DroidBench exercises many difficult flows, including flows through fields and method calls, as well as Android-specific flows. We compare with four other taint analysis tools – AppScan Source [52], Fortify SCA [53], and FlowDroid [47], using the results presented by Fritz et al. [47]. Table 7.11 summarizes the comparison. DroidInfer outperforms AppScan Source and Fortify SCA, which miss substantial amount of flows. The low recall contributes to the slightly higher precision reported by Fortify SCA. FlowDroid is slightly more precise than DroidInfer because it uses a flow-sensitive analysis. DroidBench tests for flow sensitivity and our analysis, which is flow-insensitive, misses those tests. In our experience with real-world apps however, flow sensitivity will not help.

7.4.2 Google Play Store

We analyzed 41 free Android apps from the official Google Play Store covering 24 categories. The majority of apps are from the Editor's Choice list.

Table 7.12: Actual flows in Google Play Store shown as Source→Sink pairs. The number in parentheses is the number of type errors reported by DroidInfer for the app. The majority of flows happen in advertising libraries such as InMobi, Millenial Media and Flurry, that are called from the apps.

Application (#type errors)	**Detected Leaks**
Real Fingerprint Scanner (5)	(dxjla)DeviceId→HttpEntity (parse)Location→Log
Banjo (4)	(tapjoy)DeviceId→Log
Flow Free: Bridges (5)	(flurry)DeviceId→Log (flurry)Location→Log
Chase Mobile (2)	Contact→Intent
MITBBS Reader (10)	(inmobi)Location→Log (millennialmedia)DeviceId→Log (flurry)Location→Log
Dictionary.com (13)	DeviceId→HttpEntity DeviceId→Log Location→Log (admarvel)Location→WebView (millennialmedia)DeviceId→Log (sessionm)DeviceId→Log (urbanairship)Location→Log
eBay (4)	Location→Log
Multi Touch Painting Demo (5)	(inmobi)Location→Log (millennialmedia)DeviceId→Log (flurry)Location→Log
ES Task Manager (1)	DeviceId→Intent
Facebook (4)	Location→Log PhoneNumber→Intent
Pool Billiards Pro (4)	(doodlemobile)DeviceId→Log (flurry)Location→HttpEntity
textPlus Free Text + Calls (19)	DeviceId→Log PhoneNumber→Intent PhoneNumber→HttpEntity PhoneNumber→Log SmsMessage→Log Contact→Log (tapjoy)DeviceId→Log (tapjoy)Location→Log (tapjoy)Location→WebView
NYTimes (4)	(medialets)Location→Log (medialets)Location→WebView
Solitaire (13)	(tapjoy)DeviceId→Log (flurry)DeviceId→Log (flurry)Location→HttpEntity (mopub)Location→Log (mopub)Location→WebView (tremorvideo)Location→Intent
Job Search (1)	Location→Log
Virtual Pet Care (2)	(mopub)Location→Log
Pandora internet radio (2)	(admarvel)Location→WebView
Priceline Hotels & Travel (7)	Contact→Intent Location→Log (AdX)DeviceId→Log (skyhookwireless)Location→Log (skyhookwireless)DeviceId→Log
Backgrounds HD Wallpapers (3)	DeviceId→HttpEntity DeviceId→WebView Contact→Log
Uber (5)	Location→Log
The Weather Channel (7)	Location→Log (sessionm)DeviceId→Log (sessionm)Location→Log
Noom Weight Loss Coach (9)	DeviceId→HttpEntity Location→HttpEntity Location→Log (mopub)Location→WebView (flurry)DeviceId→Log (flurry)Location→HttpEntity
Zillow Real Estate & Rentals (6)	DeviceId→Log PhoneNumber→WebView Location→Log (fiksu)DeviceId→Log
ZEDGE (6)	Contact→Intent (mopub)Location→Log (flurry)DeviceId→Log (flurry)Location→Log

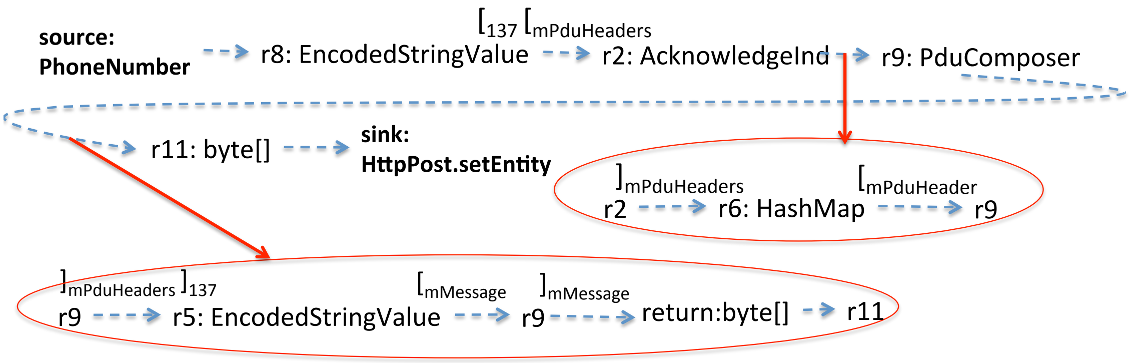


Figure 7.6: A PhoneNumber→HttpEntity flow detected by DroidInfer in the textFree app. ⇢ denotes a flow from the left side to the right side. $[_f$ denotes a write into field f and $]_f$ denotes a read from field f. The PhoneNumber is encoded in r8, which is written under key 137 in HashMap mPduHeaders, a field of r2. r2 is passed to the constructor of r9, where the mPduHeaders field is retrieved and assigned to field mPduHeader of r9. Later, the value of 137 is retrieved, appended to field mMessage of r9, which eventually is read, converted to a byte array, and sent to the HttpEntity sink. Flows r2 ⇢ r9 and r9 ⇢ r11 have corresponding method summary constraints r2 <: r9 and r9 <: r11; the flows are explained in the red ovals.

DroidInfer identified sources and sinks in 29 apps and reported 143 type errors over 25 apps. We inspected all type errors and confirmed 108 errors as actual flows from a source to a sink. This amounts to a false positive rate of 24%. To inspect, we used a tracing tool (still under development), which recorded the changes to the sets of types during type inference. Consider the leak in Figure 7.6. DroidInfer reported a type error at call r6 = tm.getLine1Number(), where the right-hand-side is inferred {tainted} (the return value of getLine1Number() is a source) and the left-hand-side is inferred {safe}. The tracing tool revealed the flow that caused the error, shown in Figure 7.6, which we confirmed by looking at the Jimple files. This flow clearly shows the power of DroidInfer — it involves dozens of method calls and field accesses, and spans half a dozen files. We identified and confirmed numerous such complex flows.

The analysis result is summarized in Table 7.12. In many cases DeviceId or Location, unencrypted, flows to Logs or the Network; we believe the purpose of these flows is tracking, in lieu of cookies.

Table 7.13: Leaks detected in Malware.

Application (#type errors)	Detected Leaks
DroidKungFu (1)	DeviceId→HttpEntity
Fakedaum (3)	SimSerialNumber→HttpEntity PhoneNumber→HttpEntity SmsMessage→HttpEntity
Godwon (5)	DeviceId→HttpEntity PhoneNumber→HttpEntity SmsMessage→HttpEntity
Jollyserv (2)	DeviceId→sendTextMessage PhoneNumber→HttpEntity
Roidsec (4)	PhoneNumber→Socket OutputStream Location→Socket OutputSteam Contact→Socket OutputStream DeviceId→Socket OutputStream

DroidInfer takes 86 seconds per app on average. It takes less than 2 minutes on 31 of the 41 apps and takes at most 5 minutes on all apps.

We ran FlowDroid, the only other publicly available taint analysis tool, on 7 apps. It threw an Out-of-memory exception on 5 of the apps. Max heap size was set to 6GB. We have confirmed with the developers of FlowDroid that it requires more than 6GB of memory.

7.4.3 Malware

We analyzed 5 known malicious apps from the contagio website [75]. Table 7.13 summarizes the analysis result. These 5 applications send out phone state (e.g. DeviceId, SimSerialNumber, and PhoneNumber), SMS messages, and/or location information through HTTP/text messages, or write into a socket.

CHAPTER 8
Related Work

8.1 Type Inference and Checking Frameworks

There are several existing pluggable type frameworks for creating and checking custom pluggable type systems: Polyglot [76], JQual [77], JastAdd [78], JavaCOP [79, 80] and the Checker Framework [18, 14].

8.1.1 Polyglot

The Polyglot framework is a compiler front end for Java. It implements an extensible compiler for Java 1.4 and a programmer who wants to implement a language extension (e.g. a pluggable type system) may extend the framework to define to the compilation process, including the Abstract Syntax Tree (AST) and semantic analysis [76]. Polyglot has been used to implement a number of Java extensions, including Jif [81], Soot [67], X10 [3], and more (see [82]).

Language extensions in Polyglot are implemented by extending the Java's grammar, type system and defining new code transformation. This process builds a compiler that reads the original program and outputs the transformed Java source code, which can be compiled by the standard Java compiler. Therefore, it is possible that the runtime behavior of the transformed program is different from the original one.

Polyglot is more suitable for building a complex language extension than implementing a pluggable type system. The runtime behavior of the transformed program by Polyglot may have changed when the language extension is added, which is not expected for pluggable types as they should not affect the runtime behavior of the original program.

8.1.2 JastAdd

JastAdd is a Java compiler that is easy to extend with new analysis as well as new language constructs [78]. The combination of object-orientation with declarative attributes and context-dependent rewrites allows highly modular specifications [83]. It can implement all language features of Java 5 as modular extensions to Java 1.4. Also, a non-null checker can be included as a pluggable component.

Programmers need to specify four principal parts to build a component: an abstract grammar defining the structure of the AST, behavior specifications defining the behavior of the AST, a context-free grammar defining how text is parsed into ASTs, and a main program for generating output based on the above three parts [78].

Building a pluggable type system on JastAdd requires a type system designer to have substantial knowledge of parsing, abstract syntax trees and symbol tables. Also, the runtime behavior of the transformed program by JastAdd may have changed as with Polyglot, which is not expected for pluggable types.

8.1.3 JQual

JQual is a framework for inferring user-defined type qualifiers in Java [77]. It is the only framework among all surveyed ones, that studies inference of qualifiers, in addition to checking. JQual is effective for source-sink type systems, for which programmers need to add annotations to the sources and sinks, and then JQual infers the intermediate annotations for the rest of the program and checks their consistency. Greenfieldboyce and Foster present two applications using JQual: inferring opaque, transparent, and $\mathsf{enum}_{\mathsf{i}}$ qualifiers in code that uses the JNI, and inferring readonly in a range of Java programs [77].

The input of JQual is a set of source files and a configuration file describing the order among the type qualifiers. Then JQual parses the source code, and generates and solves type constraints using CQual's back-end [84]. Users can choose whether to enable field sensitivity, context sensitivity, both, or neither when they run JQual. However, it is not scalable in these modes according to the authors. Artzi et al.'s evaluation confirms this [85]. In field-insensitive mode, JQual suffers from the problem that the method receiver has to be mutable when the method reads a mutable field, even if the method itself does not mutate any program state. In contrast, our analysis is scalable and may even scale better than Javarifier. JQual requires source code for all necessary classes because it is a whole-program analysis, while our analysis is modular and can analyze any given set of classes.

JQual focuses more on type inference than type-checking for pluggable types. It focuses on type systems containing a single type qualifier that induces either a supertype or a subtype of the unqualified type [18]. In contrast, our framework works not only with context-sensitive source-sink type systems, but with other complex systems such as Ownership Types and Universe Types.

8.1.4 JavaCOP

JavaCOP is a program constraint system for implementing pluggable type systems for Java [79, 80]. User-defined typing constraints are written in a declarative and expressive rule language. A pluggable type system is implemented as a set of rules, which constrain programs via the AST representation [80]. JavaCOP then translates these rules into regular Java code, which enforces the rules during a depth-first traversal of the AST.

The following rule from [79] shows how JavaCOP declaratively enforces the right-hand-side expression of each assignment to be nonnull if the left-hand-side is nonnull.

```
rule checkNonNull(Assign a) {
  where(requiresNonNull(a.lhs)) {
    require(definitelyNotNull(a.rhs)):
      error(a,"Possible null assignment" + " to @NonNull");
  }
}
```

The rule relies on two user-defined predicates. The requiresNonNull predicate checks that the given variable or field was declared with the @NonNull attribute. The definitelyNotNull predicate inspects the given expression to determine if it is definitely nonnull.

JavaCOP is able to express a number of interesting type systems, including confined types [12], scoped types [13], types for race detection [86], nonnull types, types for immutability [6], and generic ownership types [87]. Several style and convention checkers, including an EJB3.0 verifier, as well as various code pattern detectors and metrics tools are also implemented in JavaCOP [79]. In contrast, our framework focuses on type inference for context-sensitive pluggable types.

8.1.5 Checker Framework

The Checker Framework supports writing and checking pluggable type systems for Java [18, 14]. It includes the syntax in Java 8 [2] for expressing type qualifiers. A type system designer can define new type qualifiers and their semantics in a declarative and/or procedural manner. A type checker is created by the Checker Framework as a compiler plug-in. It is well-integrated with the Java language and toolset [18].

Implementing a type system requires four components:

(1) **Type qualifiers and hierarchy**. A type system designer needs to provide all qualifiers and their subytyping relation for the type system.

(2) **Type introduction rules**. Default type for a variable needs to be specified. If the variable is not annotated by the programmer, then it takes the default type qualifier.

(3) **Type rules.** The designer needs to write the typing rules for checking correctness.

(4) **Interface to the compiler**. The compiler interface indicates which annotations are part of the type system, the checker-specific compiler command-line options, etc.

8.2 Reference Immutability Systems

We begin by comparison of ReIm with Javari [6] and its inference tool Javarifier [16], which represent the state-of-the-art in reference immutability. Although the type systems have similarities, they also differ in important points of design and implementation. The corresponding inference tools implement substantially different inference algorithms. Section 8.2.1 compares ReIm with Javari, and Section 8.2.2 compares our inference approach, ReImInfer, with Javarifier. Section 8.2.3 discusses related work on purity inference, and Section 8.2.4 discusses other related work.

8.2.1 Comparison with Javari

There are two essential differences between ReIm and Javari [6]. First, Javari allows programmers to exclude fields from the abstract state by designating fields as assignable or mutable. Such a field may be assigned or mutated even through a readonly reference. An example is a field used for caching (e.g., hashCode) — modifying it should not be considered mutation from the client's point of view. As expected however, this expressive power complicates Javari: to prevent converting an immutable reference to a mutable reference, Javari requires the access to an assignable field through a readonly reference, to have different mutabilities depending on whether it is an l-value or an r-value of an assignment expression. ReIm does not allow assignable or mutable fields and therefore it is less expressive but simpler. This decision is motivated by our intended application: purity inference. Including assignable and assignable for fields in the type system would have complicated purity inference. In addition, the maximal typing would not type-check if we allowed mutable fields.

Second, Javari treats generics and arrays differently. Javari permits annotating the type arguments when instantiating a parametric class: a programmer can express designs such as "readonly list of readonly elements", "readonly list of mutable elements", "mutable list of readonly elements", and "mutable list of mutable elements". ReIm does not support annotations on the type arguments when instantiating a parametric class, and can express only "readonly list of readonly elements" and "mutable list of mutable elements" (which it uses to approximate the two inexpressible designs). The difference between the two approaches is illustrated by the following example:

```
void m(List<Date> lst2) {
  lst2.get(0).setHours(1);
}
```

Here Javari's inference tool (Javarifier) infers that reference lst2 is of type readonly List<mutable Date>. ReImInfer annotates lst2 as mutable List<Date>. (Javarifier does not have an option to make it prefer the solution mutable List<mutable Date> over readonly List<mutable Date>.) Again, the primary motivation for the decision about ReIm's simpler design is the application we had in mind: purity inference.

Purity is a single bit that summarizes whether any reachable datum may be modified, and finer-grained information is not of use when computing whether a method is pure.

Arrays are treated similarly to generics in Javari and its inference tool. In the following code b would be annotated as mutable Date readonly [].

```
void m(Date[] b) {
  b[0].setHours(2);
}
```

Again, Javari and Javarifier permit a programmer to give the array and its elements either the same or different mutability annotations. ReIm and ReImInfer enforce that the array and its elements have the same mutability annotation, so the array reference b would be inferred as mutable Date mutable [] due to the mutation of element 0.

One might imagine inferring method purity from Javarifier's output, as follows: a method is pure if all the mutabilites of its formal parameters and static variables, and their type arguments and array elements, are readonly. This approach is sound but can be unnecessarily conservative, in certain circumstances. A concrete example is when the type argument is not part of the state of the object but is mutated. Consider the following example:

```
class A<T> {
  T id(T p) { return p; }
}

void m(A<Date> x) {
  Date d = x.id(new Date());
  d.setHours(0);
}
```

Here Javarifier infers that x is of type readonly A<mutable Date>. Using the proposed approach, method m would be conservatively marked as non-pure. By contrast, ReImInfer annotates x as readonly, so m is inferred to be pure.

8.2.2 Comparison with Javarifier

Our inference approach is comparable to Javarifier, the inference tool of Javari. Both tools use flow-insensitive and context-sensitive analysis and solve constraints generated during type-based analysis. There are three substantial differences between the tools.

The most significant difference is in the context-sensitive handling of methods. The main idea of Javarifier is to create two context copies for each method that returns a reference, one copy for the case when the left-hand-side of the call assignment is mutable, and another copy for the case when the left-hand-side is readonly. As a result, Javarifier doubles the total number of method-local references, including local variables, return values, formal parameters and implicit parameters this. It also doubles the number of constraints. In contrast, our inference uses polyread and viewpoint adaptation, which efficiently captures and propagates dependences from parameters to return values in the callee, to the caller. For example, in m() { x = this.f; y = x.g; return y; }, the polyread of the return value is propagated to implicit parameter this; the dependence is transferred to the callers when viewpoint adaptation is applied at the call sites of m.

Second, Javarifier and ReImInfer have different constraint resolution approaches. Javarifier computes graph reachability over the constraint graph. Its duplication of nodes in its constraint graph correctly handles context sensitivity. In contrast, ReImInfer uses fixpoint iteration on the set-based solution and outputs the final typing based on the preference ranking over the qualifiers.

Third, Javarifier is based on Soot [67] while ReImInfer is based on the Checker Framework (CF), which did not yet exist when Javarifier was developed. Javari's type-checker is completely separate code from Javarifier, and Javarifier also requires an additional utility to map the inference result back to the source code in order to do type-checking. In total, Javari and Javarifier depend on three tools: Soot, the annotation utility, and the Checker Framework. In contrast, ReImInfer and the type-checker require only the Checker Framework and the annotation utility. These differences contribute to the usability of ReImInfer.

We conjecture that viewpoint adaptation, the constraint resolution approach, and the better infrastructure in the CF, contribute to the better scalability of ReImInfer compared to Javarifier.

8.2.3 Purity

Sălcianu and Rinard present a Java Pointer and Purity Analysis tool (JPPA) for reference immutability inference and purity inference. Their analysis is built on top of a combined pointer and escape analysis. Their analysis not only infers the immutability, but also the safety for parameters, which means the abstract state referred by a safe parameter will not be exposed to externally visible heap inside the method. However, the pointer and escape analysis is more expensive. It relies on whole program analysis, which requires main, and analyzes only methods reachable from main. ReImInfer does not require the whole program and thus it can be applied to libraries. Plus, we also include a type checker for verifying the inference result, which is not available in JPPA.

JPure [71] is a modular purity system for Java. The way JPure infers method purity is not based on reference immutability inference, as our purity inference and JPPA did. Instead, it exploits two properties, *freshness* and *locality*, for purity analysis. Its modular analysis enables inferring method purity on libraries and gains efficient runtime performance.

Rountev's analysis is designed to work on incomplete programs using fragment analysis by creating an artificial main routine [88]. However, its definition of pure method is more restricted in that it disallows a pure method to create and use a temporary object.

Clausen develops Cream, an optimizer for Java bytecode using an inter-procedural side-effect analysis [23]. It infers an instruction or a collection of instructions as pure, read-only, write-only or read/write, based on which it can infer purity for methods, loops and instructions. It is a whole-program analysis which requires a main method and also, unused methods are not covered.

Other researchers also explore the dynamic notion of purity. Dallmeier develops a tool, also called JPURE, to dynamically infer pure methods for Java [89]. The analysis calculates the *set of modified objects* for each method invocation and determines impure methods by checking if they write non-local visible objects. Xu et al. use both static and dynamic approaches to analyze method purity in Java programs [90]. Their implementation supports different purity definitions that range from strong to weak. The dynamic approach depends on the runtime behavior of programs, which is totally different from our purity analysis.

8.2.4 Other Related Work on Reference Immutability

Artzi et al. present Pidasa for classifying parameter reference immutability [91, 85]. They combine dynamic analysis and static analysis in different stages, each of which refines the result from the previous stage. The resulting analysis is scalable and produces precise result. They also incorporate optional unsound heuristics for improving precision. In contrast, our analysis is entirely static and it also infers immutability types for fields and method return values. It is unclear how their analysis handles polymorphism of methods.

The IGJ [22] and OIGJ [92] type systems support both reference immutability (a la Javari and ReIm) and also object immutability. Concurrent work by Haack et al. [93] also supports object immutability.

Porat et al. [94] present an analysis that detects immutable static fields and also addresses sealing/encapsulation. Their analysis is context-insensitive and libraries are not analyzed. Liu and Milanova [95] describe field immutability in the context of UML. Their work incorporates limited context sensitivity, analyzes large libraries and focuses on instance fields. This work is an improvement over [95]. Immutability inference not only includes instance fields, but also local variables, return values, formal parameters, and `this` parameters. Also, this work provides a type checker to verify the correctness of the inference result.

8.3 Ownership Type Systems

Several dynamic approaches for ownership inference exist [19, 96, 97, 98]. Although a dynamic approach may produce more precise results, it is inherently unsound and incurs a significant performance overhead. Also, it is difficult to generalize a dynamic approach to different type systems. In contrast, our approach is static and can be applied to multiple type systems.

Aldrich et al. [99] present an ownership type system and a type inference algorithm. Their inference creates equality, component and instantiation constraints and solves these constraints. Our inference solves different kinds of constraints, namely subtyping and adapt constraints.

Ma and Foster [100] propose Uno, a static analysis for automatically inferring ownership, uniqueness, and other aliasing and encapsulation properties in Java. Uno infers "stricter" ownership in which an owned object can only be accessed by its owner. Our inference has a less-restrictive ownership model. Uno's inference is based on Soot and it is difficult to map the inference results back to the source code, subsequently inhibiting type-checking. Our type inference is integrated into the Checker Framework; we perform type-checking as well.

Dietl et al. [17] present a tunable static inference for Generic Universe Types (GUT). Constraints of GUT are encoded as a boolean satisfiability problem, which is solved by a weighted Max-SAT solver. The inference is tunable in the sense that programmers can direct the inference by setting different weights or partially annotating the source code. In contrast, our inference can only be tuned by accepting programmers' manual annotations. However, by defining a ranking over typings, we avoid the exponential SAT solver and manage to scale to larger programs. A detailed comparison is left as future work.

Milanova and Vitek [69] present a static dominance inference analysis, based on which they perform Ownership Type inference. Our current work is an improvement over [69]. First, it accepts manual annotations to direct the inference, while [69] does not. Second, it provides optimality guarantees, while the inference in [69] does not provide guarantees — in theory, it may end up with a solution which produces a flat ownership tree. Third, our work includes a type checker which is not available in [69], and it works on more and larger benchmarks.

Sergey and Clark [40] introduce the notion of *gradual ownership types* and a corresponding consistent-subtyping relation. Their formalism provides a static guarantee of ownership invariants for fully annotated programs, but requires dynamic checks for partially-annotated programs. Their prototype works on non-generic Java programs and they analyzed 8,200 lines of code. In contrast, our inference is static and works on Java programs of up to 110kLOC.

8.4 Information Flow Systems

8.4.1 Taint Analysis for Web Applications

The most closely related to ours is the work by Shankar et al. [49]. They present a type system for detecting string format vulnerabilities in C programs. The type system has two type qualifiers, tainted and untainted; polymorphism is not part of the core system. They include a type inference engine built on top of CQual [101]. CQual relies on dependence graphs built using points-to analysis. In contrast, SFlow and SFlowInfer handle polymorphism naturally, as it is built into the type system using the poly qualifier and viewpoint adaptation. In addition, we compose with reference immutability, thus improving precision significantly. SFlow and SFlowInfer handle reflection and frameworks seamlessly.

Tripp et al. [44] present TAJ, a points-to-based taint analysis for industrial applications. TAJ is a dataflow and points-to-based analysis. In contrast, our type-based taint analysis is modular and compositional. In order to handle Struts, TAJ treats all Action classes as entry points. In addition, it simulates the passing of all subclasses of ActionForm to Action.execute, by generating a constructor, which assigns tainted values to all fields of the subclasses. In contrast, our inference analysis handles Struts by annotating the ActionForm parameter of Action.execute as tainted. Our handling is simpler and equally precise. Finally, TAJ approximates the behavior of Java reflection APIs by synthesizing an abstract object whenever the instantiated class can be inferred. It is unclear how TAJ handles reflection when the instantiated class cannot be inferred (e.g. the argument is not a string constant). According to Sridharan et al. [45], TAJ's reflection modeling is not scalable. In contrast, our type-based analysis does not need abstract objects, and handles reflection seamlessly and safely.

Livshits and Lam [43] present a static analysis based on a scalable and precise points-to analysis. The analysis is built on top of a context-sensitive Java points-to analysis [102] based on Binary Decision Diagrams (BDDs). In contrast, our inference analysis is type-based and modular. In order to handle reflection, they look for all calls to Class.forName(s) that may return className, then find all constant strings that s may refer to, and finally augment the call graph by adding an edge from the call site of newInstance to new S(), which is represented by s. Similarly to TAJ, they handle reflection by trying to infer the value of string s at forName(s).newInstance() calls. In addition, Livshits and Lam's analysis does not handle frameworks, which are essential for web applications.

Sridharan et al. [45] present F4F, a system for taint analysis of framework-based web applications. In order to handle frameworks, F4F analyzes the application code and XML configuration files to construct a specification, which summarizes reflection and callback-driven behavior. In contrast, our analysis handles frameworks by inferring or adding annotations to sources and sinks in the frameworks, which propagate to user code through subtyping. Tripp et al. [46] present ANDROMEDA, a demand-driven analysis that improves on F4F.

Volpano et al. [103] and Myers [81] present type systems for secure information flow. These systems are substantially more complex than SFlow. They focus on type-checking and do not include type inference or include only local (intra-procedural) type inference. In contrast, SFlowInfer handles large web applications.

Snelting et al. [104], Hammer et al. [105, 106], and Giffhorn and Hammer [107] present information flow analysis based on Program Dependence Graphs (PDGs). Their analysis relies on highly precise context-sensitive dataflow and points-to analysis.

The Checker Framework [18] includes a *Tainting Checker* [108] which prevents certain kinds of trust errors. The type system of the Tainting Checker is very similar to SFlow/Integrity. However, type inference is not available in the Tainting Checker. Therefore, programmers have to explicitly annotate variables that are tainted. In contrast, programmers only need to specify the sources and the sinks, and our inference framework fills the rest or reports type errors if there are flow violations. In addition, our inference also composes with ReIm to improve the typing precision.

8.4.2 Android Malware Analysis

There is a large amount of work on Android malware analysis, both dynamic and static. We focus the discussion on static analysis.

LeakMiner [64] is a points-to based static analysis for Android. It models the Android lifecycle to handle callback methods. However, LeakMiner is context-insensitive which may lead to a number of false positives in practice. It is unclear whether LeakMiner supports inter-component communications. In contrast, DroidInfer is a context-sensitive type-based static analysis and it handles inter-component communication. LeakMiner is not publicly available therefore we could not perform a direct comparison.

SCANDAL [65] is a static analyzer that detects privacy leaks in Android Apps. It directly processes Dalvik bytecode by translating the bytecode into Dalvik Core, an intermediate language designed by the SCANDAL authors. SCANDAL is limited by a high false positive rate — the average false positive rate is about 55% according to the result in [65], excluding the *unknown paths* which make up more than half of the total paths. In contrast, DroidInfer is based on Soot and Dexpler which first transforms Dalvik bytecode into Java bytecode and then to Jimple code. DroidInfer appears to be more precise — the average false positive rate is 24%. In addition, SCANDAL is Samsung proprietary and unavailable to us for comparison.

AndroidLeaks [66] finds potential leaks of private information in Android Apps. It uses WALA to construct a context-sensitive System Dependence Graph (SDG) and a context-insensitive overlay for tracking heap dependencies in SDG. It lacks field sensitivity because it taints whole objects that have tainted data stored inside them. In contrast, DroidInfer is type-based and does not need abstraction of heap objects. In addition, DroidInfer is field-sensitive and is more precise than AndroidLeaks. AndroidLeaksis not publicly available.

CHEX [63] can automatically vet Android apps for component hijacking vulnerabilities. It models the vulnerabilities from a data-flow analysis perspective and detects possible *hijack-enabling* flows. CHEX can also detect private data leakage. To avoid the complexity of analyzing the Android framework library, CHEX models the Android framework. In contrast, DroidInfer detects private data leaks in Android Apps. And it handles Android library methods differently by making reasonable assumptions.

FlowDroid [47] is a context-, object- and flow-sensitive taint analysis for Android. It models Android's lifecycle to handle callbacks. In contrast, DroidInfer is type-based, and it handles callbacks with function subtyping. FlowDroid does not handle ICC, while DroidInfer does. Also, FlowDroid handles library calls by using manual summaries, while we make conservative assumptions for libraries. As discussed in Section 7.4, FlowDroid is incomparable to DroidInfer, because it uses a larger set of sources and sinks [109].

SCanDroid [110] focuses on ICC. It formalizes the data flows through and across components in a core calculus. Epicc [111] discovers ICC for Android apps by identifying a *specification* for every ICC source and sink, including the ICC Intent action, data type, category, etc. We plan to integrate Epicc in DroidInfer, which will provide more channels for privacy leaks.

8.5 Other Related Work

Work on introducing generics to Java [112, 113] solves similar challenges, because leaving every type as raw is a legal typing, but a useless one that expresses no design intent and detects no coding errors. In contrast to our work, Donovan et al. [112] use heuristics to find desirable solutions and their inference requires a pointer analysis. Kieżun et al. [113] make use of type constraints to ensure behavior preservation. They also use heuristics, otherwise user's input is required.

Our algorithm for computing the set-based solution (Section 2.3) is similar to the algorithm used by Tip et al. [113, 114]. Both algorithms start with sets containing all possible answers and iteratively remove elements that are inconsistent with the typing rules. Our work differs as we introduce a ranking over valid typings and use the ranking to guide the automatic inference towards a final "best" typing.

Our work, as well as [114], falls in the category of type-based and constraint-based analysis, originally proposed by Palsberg and Schwartzbach [115].

CHAPTER 9
Conclusion and Future Work

Mandatory typing does not always guarantee that “well-typed” programs will not go wrong, because most built-in mandatory type systems do not enforce some important program properties for the languages. On the other hand, pluggable types enforces many important program properties. However, most pluggable type systems require annotations in the source code and the annotation burden may inhibit the adoption of pluggable types in practice. In this thesis, we have proposed an inference and checking framework, for specifying, inferring, and checking of context-sensitive pluggable types.

9.1 Inference and Checking Framework

We have presented an inference and checking framework for context-sensitive pluggable types. By supplying five framework parameters: (1) type qualifiers, (2) subtyping relation, (3) viewpoint adaptation rules, (4) context of adaptation, and (5) additional constraints, programmers can instantiate the framework’s unified typing rules into concrete ones for a specific type system. The framework then takes as input an unannotated or a partially annotated program, and infers the most desirable typing (according to the objective function defined by the programmer), and verifies the correctness of the typing. As a result, programmers do not need to annotate every variable in their program. Instead, they can choose to annotate only variables they care about, or none, and the framework infers the rest. Programmers can use the framework to not only infer and plug existing type systems, but also build new type systems.

9.2 Instantiations of Pluggable Type Systems

In this thesis, we have presented five instantiations of the inference and checking framework, including the reference immutability type system ReIm, the classical Ownership Types, Universe Types, the information flow integrity type system

SFlow/Integrity, and its dual confidentiality type system SFlow/Confidentiality. Note that, other than the type systems presented in this thesis, we have also instantiated the inference and checking framework with AJ [9], EnerJ [11], and more.

For each of these instantiated type systems, we have evaluated the corresponding type inference and type checking extensively by a number of small-to-large Java and Android applications. The evaluations show that our unified type inference approach is general enough to handle different type systems, but also customizable to handle specific features of the type systems. The inference results indicate that our inference approach is both scalable and precise.

9.3 Future Work

One direction of future work is to instantiate the framework with more interesting type systems, such as *nonnull references* [4], *interning types* [14], and more. These type systems are very useful, as they can prevent unforeseen runtime errors. However, extensive annotation burden has inhibited their practical adoption. We believe that with our inference and checking framework, programmers can benefit from these type systems with little or no annotation burden.

Another direction of future work is to further investigate the theory behind the framework. For example, the problem of extracting the “best” valid typing from the set-based solution needs further investigation. Currently we use the maximal typing with the extension of method summary constraints. In the future, we may want to study other extraction techniques for efficiently extracting the “best” typing.

Finally, we want to integrate our inference analysis with the Checker Framework [18] as a plug-in. With the release of Java 8, more and more programmers will start writing checkers for their programmers using the Checker Framework. However, there is no built-in type inference in the Checker Framework and programmers may have to write a sizable amount of annotations. Since our type inference is built upon the Checker Framework, it should be easily adapted into the Checker Framework as a plug-in.

BIBLIOGRAPHY

[1] G. Bracha, “Pluggable type systems,” in *Proc. Workshop Revival of Dynamic Languages*, Vancouver, Canada, 2004, pp. 1–6.

[2] M. D. Ernst. (2014, Jan. 11). *Type Annotations (JSR 308) and the Checker Framework* [Online]. Available: http://types.cs.washington.edu/jsr308/ (Retrieved on Mar. 27, 2014).

[3] T. Wrigstad *et al.*, “Integrating typed and untyped code in a scripting language,” in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Madrid, Spain, 2010, pp. 377–388.

[4] M. Fähndrich and K. R. M. Leino, “Declaring and checking non-null types in an object-oriented language,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, 2003, pp. 302–312.

[5] J. Boyland *et al.*, “Capabilities for sharing: A generalisation of uniqueness and Read-Only,” in *Proc. European Conf. Object-Oriented Programming*, Budapest, Hungary, 2001, pp. 2–27.

[6] M. S. Tschantz and M. D. Ernst, “Javari: Adding reference immutability to Java,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, 2005, pp. 211–230.

[7] D. Clarke *et al.*, “Ownership types for flexible alias protection,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, 1998, pp. 48–64.

[8] W. Dietl and P. Müller, “Universes: Lightweight ownership for JML,” *J. of Object Technology*, vol. 4, no. 8, pp. 5–32, Oct. 2005.

[9] M. Vaziri *et al.*, “A type system for data-centric synchronization,” in *Proc. European Conf. Object-Oriented Programming*, Maribor, Slovenia, 2010, pp. 304–328.

[10] J. Dolby *et al.*, “A data-centric approach to synchronization,” *ACM Trans. Programming Languages and Syst.*, vol. 34, no. 1, pp. 1–48, Apr. 2012.

[11] A. Sampson *et al.*, “EnerJ: Approximate data types for safe and general low-power computation,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, San Jose, CA, 2011, pp. 164–174.

[12] J. Vitek and B. Bokowski, "Confined types," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO, 1999, pp. 82–96.

[13] T. Zhao *et al.*, "Scoped types for real-time Java," in *Proc. IEEE Real-Time System Symp.*, Lisbon, Portugal, 2004, pp. 241–251.

[14] W. Dietl *et al.*, "Building and using pluggable type-checkers," in *Proc. Int. Conf. Software Engineering*, Honolulu, HI, 2011, pp. 681–690.

[15] W. Huang *et al.*, "ReIm & ReImInfer: Checking and inference of reference immutability and method purity," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Tucson, AZ, 2012, pp. 879–896.

[16] J. Quinonez *et al.*, "Inference of reference immutability," in *Proc. European Conf. Object-Oriented Programming*, Paphos, Cyprus, 2008, pp. 616–641.

[17] W. Dietl *et al.*, "Tunable static inference for generic universe types," in *Proc. European Conf. Object-Oriented Programming*, Lancaster, UK, 2011, pp. 333–357.

[18] M. M. Papi *et al.*, "Practical pluggable types for Java," in *Proc. Int. Symp. Software Testing and Analysis*, Seattle, WA, 2008, pp. 201–212.

[19] W. Dietl and P. Müller, "Runtime universe type inference," in *Proc. Int. Workshop Aliasing, Confinement and Ownership in Object-Oriented Programming*, Berlin, Germany, 2007, pp. 72–80.

[20] D. Cunningham *et al.*, "Universe types for topology and encapsulation," in *Proc. Formal Methods for Components and Objects*, Sophia Antipolis, France, 2008, pp. 72–112.

[21] F. Nielson *et al.*, *Principles of Program Analysis.* Berlin, Germany: Springer-Verlag, 1999.

[22] Y. Zibin *et al.*, "Object and reference immutability using Java generics," in *Proc. Joint Meeting of European Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering*, Dubrovnik, Croatia, 2007, pp. 75–84.

[23] L. R. Clausen, "A Java bytecode optimizer using side-effect analysis," *Concurrency and Computation: Practice and Experience*, vol. 9, no. 11, pp. 1031–1045, Nov. 1997.

[24] A. Le *et al.*, "Using inter-procedural side-effect information in JIT optimizations," in *Proc. Int. Conf. Compiler Construction*, Edinburgh, UK, 2005, pp. 287–304.

[25] J. Zhao *et al.*, “Pure method analysis within Jikes RVM,” in *Proc. Int. Workshop Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, Paphos, Cyprus, 2008, pp. 1–8.

[26] O. Tkachuk and M. B. Dwyer, “Adapting side effects analysis for modular program model checking,” in *Proc. Joint Meeting of European Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering*, Helsinki, Finland, 2003, pp. 188–197.

[27] W. Huang *et al.*, “Inference and checking of object ownership,” in *Proc. European Conf. Object-Oriented Programming*, Beijing, China, 2012, pp. 181–206.

[28] A. Heydon *et al.*, “Caching function calls using precise dependencies,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Vancouver, Canada, 2000, pp. 311–320.

[29] A. Sălcianu and M. Rinard, “Purity and side effect analysis for Java programs,” in *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation*, Paris, France, 2005, pp. 199–215.

[30] E. C. Chan *et al.*, “Promises: Limited specifications for analysis and manipulation,” in *Proc. Int. Conf. Software Engineering*, Kyoto, Japan, 1998, pp. 167–176.

[31] P. Müller, *Modular specification and verification of object-oriented programs.* Berlin, Germany: Springer-Verlag, 2002.

[32] K. R. M. Leino, “Data groups: Specifying the modification of extended state,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, 1998, pp. 144–153.

[33] J. Hogg *et al.*, “The Geneva convention on the treatment of object aliasing,” *ACM SIGPLAN OOPS Messenger*, vol. 3, no. 2, pp. 11–16, Apr. 1992.

[34] C. Boyapati *et al.*, “Ownership types for safe programming: Preventing data races and deadlocks,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002, pp. 211–230.

[35] A. Milanova and W. Huang, “Static object race detection,” in *Proc. Asian Conf. Programming Languages and Systems*, Kenting, Taiwan, 2011, pp. 255–271.

[36] S. Negara *et al.*, “Inferring ownership transfer for efficient message passing,” in *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, San Antonio, TX, 2011, pp. 81–90.

[37] K. R. M. Leino and P. Müller, "Object invariants in dynamic contexts," in *Proc. European Conf. Object-Oriented Programming*, Oslo, Norway, 2004, pp. 491–515.

[38] C. Boyapati *et al.*, "Ownership types for object encapsulation," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, New Orleans, LA, 2003, pp. 213–223.

[39] J. Noble *et al.*, "Flexible alias protection," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, 1998, pp. 1–28.

[40] I. Sergey and D. Clarke, "Gradual ownership types," in *Proc. European Symp. Programming*, Tallinn, Estonia, 2012, pp. 579–599.

[41] W. Huang *et al.*, "Type-based taint analysis for Java web application," in *Int. Conf. Fundamental Approaches to Software Engineering*, Grenoble, France, 2014, pp. 140–154.

[42] OWASP. (2014, Mar. 21). *OWASP Top Ten Project* [Online]. Available: https://www.owasp.org/index.php/Top_Ten (Retrieved on Mar. 21, 2014).

[43] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proc. USENIX Security Symp.*, Baltimore, MD, 2005, pp. 271–286.

[44] O. Tripp *et al.*, "TAJ : Effective taint analysis of web application," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Dublin, Ireland, 2009, pp. 87–97.

[45] M. Sridharan *et al.*, "F4F : Taint analysis of framework-based web application," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, 2011, pp. 1053–1068.

[46] O. Tripp *et al.*, "ANDROMEDA: Accurate and scalable security analysis of web applications," in *Int. Conf. Fundamental Approaches to Software Engineering*, Rome, Italy, 2013, pp. 210–225.

[47] C. Fritz *et al.*, "Highly precise taint analysis for Android application," Dept. Comput. Sci., Tech. Univ. of Darmstadt, Darmstadt, Germany, Tech. Rep. TUD-CS-2013-0113, 2013.

[48] A. C. Myers *et al.*, "Parameterized types for Java," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Paris, France, 1997, pp. 132–145.

[49] U. Shankar *et al.*, "Detecting format string vulnerabilities with type qualifiers," in *Proc. USENIX Security Symp.*, Washington, D.C., 2001, pp. 201–220.

[50] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, St. Petersburg Beach, FL, 1996, pp. 32–41.

[51] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, Dept. Comput. Sci., Univ. of Copenhagen, Copenhagen, Denmark, 1994.

[52] IBM. (2014, Mar. 2). *IBM Security AppScan* [Online]. Available: http://www-03.ibm.com/software/products/en/appscan/ (Retrieved on Mar. 26, 2014).

[53] HP. (2014, Mar. 14). *Application Security: HP Fortify end to end software security for the new style of IT* [Online]. Available: http://www8.hp.com/us/en/software-solutions/application-security/ (Retrieved on Mar. 23, 2014).

[54] A. Milanova and W. Huang, "Dataflow and type-based formulations for reference immutability," in *Proc. Workshop Formal Techniques for Java-like Programs*, Montpellier, France, 2013, pp. 5:1–5:7.

[55] A. Milanova *et al.*, "Parameterized object sensitivity for points-to analysis for Java," *ACM Trans. Software Eng. and Methodology*, vol. 14, no. 1, pp. 1–41, Jan. 2005.

[56] Engadget. (2013, Oct. 31). *Android tops 81 percent of smartphone market share in Q3* [Online]. Available: http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-phone-share/ (Retrieved on Mar. 23, 2014).

[57] F-Secure. (2013, Oct. 13). *Mobile Threat Report* [Online]. Available: http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf (Retrieved on Mar. 26, 2014).

[58] Google. (2012, Feb. 2). *Android and Security* [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html (Retrieved on Mar. 26, 2014).

[59] W. Enck *et al.*, "TaintDroid : An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX Conf. Operating System Design and Implementation*, Vancouver, Canada, 2010, pp. 1–6.

[60] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proc. USENIX Security Symp.*, Bellevue, WA, 2012, pp. 569–584.

[61] A. Reina *et al.*, “A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors,” in *Proc. European Workshop System Security*, Prague, Czech Republic, 2013, pp. 1–6.

[62] R. Xu *et al.*, “Aurasium: Practical policy enforcement for Android application,” in *Proc. USENIX Security Symp.*, Bellevue, WA, 2012, pp. 539–552.

[63] L. Lu *et al.*, “CHEX : Statically vetting Android apps for component hijacking vulnerabilities categories and subject descriptors,” in *Proc. ACM Conf. Computer and Communications Security*, Raleigh, NC, 2012, pp. 229–240.

[64] Z. Yang and M. Yang, “LeakMiner: Detect information leakage on Android with static taint analysis,” in *Proc. World Congr. Software Engineering*, Wuhan, China, 2012, pp. 101–104.

[65] J. Kim *et al.*, “SCANDAL: Static analyzer for detecting privacy leaks in Android application,” in *Proc. Mobile Security Technologies*, San Francisco, CA, 2012, pp. 1–10.

[66] C. Gibler *et al.*, “AndroidLeaks: Automatically detecting potential privacy leaks in Android application on a large scale,” in *Proc. Int. Conf. Trust and Trustworthy Computing*, Vienna, Austria, 2012, pp. 291–307.

[67] R. Vallée-Rai *et al.*, “Soot: A Java bytecode optimization framework,” in *Proc. Conf. Centre for Advanced Studies on Collaborative Research*, Mississauga, Canada, 1999, pp. 13–23.

[68] A. Bartel *et al.*, “Dexpler: Converting Android dalvik bytecode to jimple for static analysis with Soot,” in *Proc. ACM SIGPLAN Int. Workshop State of the Art in Java Program Analysis*, Beijing, China, 2012, pp. 27–38.

[69] A. Milanova and J. Vitek, “Static dominance inference,” in *Proc. Int. Conf. Objects, Models, Components and Patterns*, Zurich, Switzerland, 2011, pp. 211–227.

[70] M. Barnett *et al.*, “99.44% pure: Useful abstractions in specifications,” in *Proc. Workshop Formal Techniques for Java-like Programs*, Oslo, Norway, 2004, pp. 11–19.

[71] D. Pearce, “JPure: A modular purity system for Java,” in *Proc. Int. Conf. Compiler Construction*, Saarbrucken, Germany, 2011, pp. 104–123.

[72] W. Huang and A. Milanova, “Towards effective inference and checking of ownership types,” in *Proc. Int. Workshop Aliasing, Confinement and Ownership in Object-Oriented Programming*, Lancaster, UK, 2011, pp. 1–11.

[73] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in Java applications with static analysis,” Dept. Comput. Sci., Stanford Univ., Stanford, CA, Tech. Rep. SU-CS-2005-01, 2005.

[74] Google. (2014, Mar. 24). *Google Play Store* [Online]. Available: https://play.google.com (Retrieved on Mar. 24, 2014).

[75] Mila. (2014, Mar. 22). *Contagio mobile* [Online]. Available: http://contagiominidump.blogspot.com (Retrieved on Mar. 22, 2014).

[76] N. Nystrom *et al.*, “Polyglot: An extensible compiler framework for Java,” in *Proc. Int. Conf. Compiler Construction*, Warsaw, Poland, 2003, pp. 138–152.

[77] D. Greenfieldboyce and J. S. Foster, “Type qualifier inference for java,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Canada, 2007, pp. 321–336.

[78] T. Ekman and G. Hedin, “The Jastadd extensible Java compiler,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Canada, 2007, pp. 1–18.

[79] C. Andreae *et al.*, “A framework for implementing pluggable type systems,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, 2006, pp. 57–74.

[80] S. Markstrum *et al.*, “JavaCOP: Declarative pluggable types for Java,” *ACM Trans. Programming Languages and Syst.*, vol. 32, no. 2, pp. 1–37, Jan. 2010.

[81] A. C. Myers, “JFlow: Practical mostly-static information flow control,” in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, San Antonio, TX, 1999, pp. 228–241.

[82] N. Nystrom *et al.* (2013, June 21). *Polyglot: A compiler front end framework for building Java language extensions* [Online]. Available: http://www.cs.cornell.edu/Projects/polyglot/ (Retrieved on Mar. 24, 2014).

[83] T. Ekman and G. Hedin, “Pluggable checking and inferencing of nonnull types for Java,” *J. of Object Technology*, vol. 6, no. 9, pp. 455–475, Oct. 2007.

[84] J. S. Foster *et al.*, “Flow-insensitive type qualifiers,” *ACM Trans. Programming Languages and Syst.*, vol. 28, no. 6, pp. 1035–1087, Nov. 2006.

[85] S. Artzi *et al.*, “Parameter reference immutability: Formal definition, inference tool, and comparison,” *Automated Software Eng.*, vol. 16, no. 1, pp. 145–192, Dec. 2009.

[86] C. Flanagan and S. N. Freund, “Type-based race detection for Java,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Vancouver, Canada, 2000, pp. 219–232.

[87] A. Potanin *et al.*, “Generic ownership for generic Java,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, 2006, pp. 311–324.

[88] A. Rountev, “Precise identification of side-effect-free methods in Java,” in *Proc. IEEE Int. Conf. Software Maintenance*, Chicago, IL, 2004, pp. 82–91.

[89] V. Dallmeier, “Static vs. dynamic purity analysis,” Dept. Comput. Sci., Saarland Univ., Saarbrucken, Germany, Tech. Rep. SU-CS-2013-012, 2007.

[90] H. Xu *et al.*, “Dynamic purity analysis for Java programs,” in *Proc. Workshop Program Analysis for Software Tools and Engineering*, San Diego, CA, 2007, pp. 75–82.

[91] S. Artzi *et al.*, “Combined static and dynamic mutability analysis,” in *Proc. Int. Conf. Automated Software Engineering*, Atlanta, GA, 2007, pp. 104–113.

[92] Y. Zibin *et al.*, “Ownership and immutability in generic Java,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Reno, NV, 2010, pp. 598–617.

[93] C. Haack *et al.*, “Immutable objects for a Java-like language,” in *Proc. European Symp. Programming*, Braga, Portugal, 2007, pp. 347–362.

[94] S. Porat *et al.*, “Automatic detection of immutable fields in Java,” in *Proc. Centre for Advanced Studies on Collaborative Research*, Mississauga, Canada, 2000, pp. 10–24.

[95] Y. Liu and A. Milanova, “Ownership and immutability inference for UML-based object access control,” in *Proc. Int. Conf. Software Engineering*, Minneapolis, MN, 2007, pp. 323–332.

[96] N. Mitchell, “The runtime structure of object ownership,” in *Proc. European Conf. Object-Oriented Programming*, Nantes, France, 2006, pp. 74–98.

[97] A. Potanin *et al.*, “Checking ownership and confinement,” *Concurrency and Computation: Practice and Experience*, vol. 16, no. 7, pp. 671–687, Jun. 2004.

[98] M. Vechev *et al.*, “PHALANX: Parallel checking of expressive heap assertions,” in *Proc. Int. Symp. Memory Management*, Toronto, Canada, 2010, pp. 41–50.

[99] J. Aldrich *et al.*, “Alias annotations for program understanding,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002, pp. 311–330.

[100] K.-K. Ma and J. S. Foster, “Inferring aliasing and encapsulation properties for java,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Canada, 2007, pp. 423–440.

[101] J. S. Foster *et al.*, "A theory of type qualifiers," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Atlanta, GA, 1999, pp. 192–203.

[102] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Washington, D.C., 2004, pp. 131–144.

[103] D. Volpano *et al.*, "A sound type system for secure flow analysis," *J. of Comput. Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.

[104] G. Snelting *et al.*, "Efficient path conditions in dependence graphs for software safety analysis," *ACM Trans. Software Eng. and Methodology*, vol. 15, no. 4, pp. 410–457, Oct. 2006.

[105] C. Hammer *et al.*, "Intransitive noninterference in dependence graphs," in *Proc. Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, 2006, pp. 119–128.

[106] C. Hammer *et al.*, "Static path conditions for Java," in *Proc. Workshop Programming Languages and Analysis for Security*, Tucson, AZ, 2008, pp. 57–66.

[107] D. Giffhorn and C. Hammer, "Precise analysis of Java programs using JOANA," in *Proc. IEEE Int. Workshop Source Code Analysis and Manipulation*, Beijing, China, 2008, pp. 267–268.

[108] U. of Washington. (2014, Mar. 1). *The Checker Framework Manual: Custom pluggable types for Java* [Online]. Available: http://types.cs.washington.edu/checker-framework/current/checkers-manual.html (Retrieved on Mar. 20, 2014).

[109] S. Rasthofer *et al.*, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. Symp. Network and Distributed System Security*, San Diego, CA, 2014, pp. 1–15.

[110] A. P. Fuchs *et al.*, "SCanDroid : Automated security certification of Android application," unpublished.

[111] D. Octeau *et al.*, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. USENIX Security Symp.*, Washington, D.C., 2013, pp. 543–558.

[112] A. Donovan *et al.*, "Converting Java programs to use generic libraries," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, 2004, pp. 15–34.

[113] A. Kieżun *et al.*, "Refactoring for parameterizing Java classes," in *Proc. Int. Conf. Software Engineering*, Minneapolis, MN, 2007, pp. 437–446.

[114] F. Tip *et al.*, "Refactoring using type constraints," *ACM Trans. Programming Languages and Syst.*, vol. 33, no. 3, pp. 9:1–9:47, May 2011.

[115] J. Palsberg and M. I. Schwartzbach, *Object-Oriented Type System.* Hoboken, NJ: Wiley, 1994.