PRACTICAL STATIC ANALYSIS FRAMEWORK FOR INFERENCE OF SECURITY-RELATED PROGRAM PROPERTIES

By

Yin Liu

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

Stephen J Fink, Member

Mukkai Krishnamoorthy, Member

David Musser, Member

Carlos Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

February 2010
(For Graduation May 2010)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank all people who have helped and inspired me during my doctoral study.

I especially want to express my sincere gratitude to my advisor and mentor, Professor Ana Milanova, for her continuous support and help of my Ph.D study and research. Her immense knowledge and enthusiasm in research had always motivated me. She always believed in my ability. She was always there to listen and to give advice. She taught me how to research and how to write. Thanks to Ana, the research life was joyful and rewarding for me.

Besides my advisor, I would like to thank the rest of my thesis committee for their encouragement, insightful comments, and inspirations.

I am forever indebted to my parents for their unflagging love and support throughout my life; this dissertation is simply impossible without them. Finally, my thanks to my dearest husband, Zhongyi Xie, for his understanding, endless patience and encouragement when it was most required.

ABSTRACT

For the software quality and security concerns, it is important to reason about security-related program securities. We present a static analysis framework for inference of security-related program properties. Within this framework we infer ownership, immutability and information flow for the protection of object access, data confidentiality and integrity. We propose runtime models that capture these properties. We design and implement ownership, immutability and information flow inference analyses for Java. These analyses reveal information about object access and information flow in the program, and may help uncover serious vulnerabilities.

To evaluate the framework, an empirical investigation is performed on a set of Java components, and a set of small-to-large Java programs. The results indicate that the analyses are practical and precise. Therefore, the analyses can be integrated in program comprehension tools that support effective reasoning about software security and software quality.

The usage of the inferences is illustrated by several applications of the framework. Ownership analysis is applied on reasoning about shared objects in open concurrent Java programs. Three structural patterns for object sharing are identified: the shared objects are categorized as *central*, *owned* or *distributed*. We argue that these patterns facilitate the understanding of concurrent programs. The experiments on several medium-to-large Java programs reveal the structure of sharing in real-world Java programs.

The usage of the static information flow analysis is illustrated with three applications. The first application of information flow analysis is security violation detection. We perform experiments on a set of Java web applications and the experiments show that the information flow analysis effectively detects security violations. The second application is type inference. Our experiments on the Java web applications show that our flow analysis successfully infers security types. The last application studies the effect of thread-shared variables on thread-local variables. Our experiments on a set of multi-thread programs show that most of the

thread-local variables are affected by the thread-shared variables.

CHAPTER 1
Introduction

A central and critical aspect of the computer security problem is software security. Software, as a resource itself, contains and controls data and other resources. Thus it must be designed and implemented with appropriate security properties, to protect those resources. However, current languages such as Java do not provide effective mechanisms for reasoning about program security properties. Then software implementations may have violations against these security properties. Specifically, there may exist unintended object access through aliasing, leading to unintended representation exposure or mutation of objects; or there may exist unintended information flow that leaks or tampers sensitive data, violating the data confidentiality and integrity requirements. These unintended violations may cause serious quality and security concerns, such as the known Signers bug in Java 1.1 [1].

It is important to study mechanisms for reasoning about program security properties. This problem has received considerable attention. The vast majority of work falls into two categories: (1) *dynamic, instrumentation-based* approaches such as tainting, which labels data, propagates and checks the labels during execution through suitable instrumentation; and (2) *static, language-based* approaches such as type systems, which extends type system with security types and verifies consistence of security through type checking. The disadvantage of the dynamic approaches is that they typically incur significant run-time overhead; the disadvantage of the static, language-based approaches is that they typically require non-trivial type annotations; thus, it might be difficult to adopt these approaches in practice.

On the other hand, reasoning about security properties with static analysis, which works before program execution and on current languages without the burden of annotations, has received considerably less attention. This is surprising, given that static analysis has great potential to be useful in practice: it does not incur run-time overhead, and it does not require annotations.

We propose a new general-purpose framework based on static analysis, for

reasoning about security-related program properties. Specifically, we reason about *ownership*, *immutability* and *information flow* for the purpose of revealing information about object access, protection of data confidentiality and data integrity in the program, and helping uncover serious vulnerabilities. Note that these are the properties we have explored so far; the framework can be easily extended with reasoning on other security-related properties.

The security-related properties at the design level could be illustrated as constraints on the well-known Unified Modeling Language (UML) class diagrams. These constraints impose requirements on the program run-time behaviors. These run-time behavior requirements are illustrated by our run-time models, which specify what the design level security constraints mean at runtime. Then our static analyses are applied on code, which automatically retrieve the implemented constraints according to the run-time models. Thus developers can check, verify, and reason about their code as needed: violations of specified constraints are revealed intuitively in the reverse-engineered UML class diagram.

The framework works not only on complete programs, but on incomplete programs (i.e., software components) as well. This is an important feature because often we are interested in analysis of software components which answers the following question: given a set of interacting classes (e.g., a secure server-side component), could there be compromising object access or information flow for some client of these classes?

This framework for inference of security-related properties is light-weight, works directly on Java code before program execution, and does not require annotations by the programmer. It can be easily incorporated in program understanding and verification tools and help verify in a lightweight and practical manner the program security properties. The analyses we have implemented are practical and precise, and we conjecture that they can help support effective reasoning about security.

To illustrate the usage of our analysis framework, we apply our ownership inference and information flow inference on several applications. We apply ownership inference on reasoning about shared objects in open concurrent Java programs to

facilitate the understanding of concurrent programs. We apply the information flow inference on three client applications: security violation detection, type inference and a study of the effect of thread-shared variables on thread-local variables. On these three applications, we also study the impact of implicit flow versus explicit flow.

We believe that our work is a step forward towards the use of static analysis for the purposes of reasoning about program security; it may help advance the use of static analysis in tools for understanding and verification of security properties.

1.1 Contributions

The work presented in this thesis has three major contributions.

1.1.1 Framework for Inference of Security-related Properties

We present a practical general-purpose framework for inference of security properties. The security-related properties at the design level are captured by annotations on UML diagrams.

The security properties of object access control are specified by *ownership* and *immutability* constraints. *Ownership constraints* are specified on associations. An association from class A to class B can be specified as `owned` at design level; this states a requirement for ownership and no *representation exposure* at implementation level: an A object must control the B objects it references through this association.

Immutability constraints are specified on method parameters, methods and associations. A parameter p in method m can be specified as `read-only`; this states the requirement that no invocation of m modifies the heap structure rooted at the object referred by p. A method m can be specified as `read-only`; this states the requirement that no invocation of m modifies the visible state. Finally, an association from class A to class B can be specified as `read-only`; this states a requirement for immutability: an A object cannot modify the heap structure rooted at the B object it references through this association.

The security properties for data confidentiality and data integrity are speci-

fied by *information flow constraints*. *Information flow constraints* are specified on associations and fields. An association, or a field of simple type may be specified as `confidential`; this states a requirement for confidentiality: there is no information flow from that field to an untrusted client. Finally, an association, or a field of simple type may be specified as `safe`; this states a requirement for integrity: there is no information flow from an untrusted client to that field.

Our framework retrieves security properties with run-time models and static analyses. The framework could be augmented with different run-time models and corresponding analyses.

We present run-time models that capture the meanings of ownership, immutability, confidentiality and integrity at design level. These models help to understand these security properties and instruct how to identify violations against the security properties in program execution.

The definition of implementation-level ownership is based on owners-as-dominators [2, 3]—that is, all access paths to an owned object should pass through its owner. The definition of immutability requires that an enclosing object have read-only access to an enclosed immutable object—that is, the methods invoked on the enclosing object cannot change (directly, or through callees) the heap structure rooted at the immutable object.

The information flow model distinguishes two types of flow: *shallow flow* and *deep flow*. Shallow flow covers the standard notion of information flow; that is, there is shallow flow from a variable l into a variable r if changes in the value of l impact the value of r [4]. Deep flow considers flow from (and into) the object structure rooted at a reference variable l; there is deep flow from l into r if a field reachable from l flows to r (i.e., changes in the object structure rooted at l impact the value of r, without changes in the value of l).

The confidentiality property requires that there is no information flow, shallow or deep, from the protected data to any untrusted destination. The integrity property requires that there is no information flow, shallow or deep, from any untrusted source to the protected data.

With these run-time models, we propose novel static analyses that infers own-

ership, immutability, and information flows directly from the code, according to these models.

These analyses work directly on Java code and do not require annotations by the programmer. It is important to note that the analyses can work on complete programs (i.e., whole programs) as well as on incomplete programs (i.e., software components). This is an important feature because reasoning about security should be performed on software components; any realistic program understanding and verification tool should be able to work on software components and thus cannot accommodate analysis that works only on complete programs.

1.1.2 Practical and Precise Analyses Results

We have implemented the static analysis framework for both complete programs and software components. We present an empirical study on several small-to-large Java applications, and a set of Java components. In our experiments on complete programs, on average 26% of the reverse-engineered fields were determined to be `owned`, 29% fields were determined to be `read-only`. Furthermore, 43% of the examined data were determined to be `leaked`, 23% were determined to be `tampered`. We present a precision evaluation which indicates that the analyses achieve adequate precision—of the fields we checked for false positives, ownership inference did not miss any `owned` association, the immutability inference only missed 3 `read-only` association, and all except 1 identified confidentiality and integrity violations could actually happen. On benchmarks which have about 5000 to 9000 reachable analyzed methods, on a 900MHz Sun Fire 380R machine, the ownership and immutability analyses are practical, running in less than 15 minutes on all but one benchmarks which take about 44 minutes. The flow analysis including confidentiality and integrity runs within 8 minutes on all benchmarks. The experiments indicate that (i) the models capture well the meaning of ownership, immutability, confidentiality and integrity in design, and the analyses produce useful results and (ii) the analyses are precise and practical. Therefore, the analyses can be incorporated in software tools and can effectively support verification of ownership and immutability; this will lead to high quality, secure, understandable and maintainable software systems.

1.1.3 Applications of The Framework

To illustrate the usage of our analysis framework, we apply our ownership inference and information flow inference on several applications.

1.1.3.1 An Application of Ownership Analysis

We apply ownership inference on reasoning about shared objects in open concurrent Java programs — that is, incomplete Java programs (i.e. libraries) designed for use by multi-threaded clients. We propose three structural patterns for object sharing: we classify shared objects as *central*, *owned* or *distributed*. Informally, a *central* object is an object directly accessed by client threads (e.g., method $run()$ executed on a client thread, calls a method on the central object). An *owned* object is an object indirectly accessed by client threads (e.g., $run()$ executes a method m on a central object, and m in turn executes a method on the owned object). However, an owned object is "dominated" by an "owner" object, meaning (informally) that each access to that object goes through the "owner" object. A *distributed* object is indirectly accessed by client threads as well; however, a distributed object is not dominated by an "owner" object; instead, access is distributed through multiple objects. We argue that these patterns facilitate the understanding of concurrent programs. We conjecture that well-structured concurrent programs should emphasize the role of ownership - they should strive for a larger number of owned objects and for a minimal number of distributed objects. We perform experiments on several medium-to-large Java programs, which reveal the structure of sharing in real-world Java programs.

1.1.3.2 Applications of Information Flow Analysis

We illustrate the information flow analysis on three client applications. The first client application is security violation detection. We perform experiments on a suite of Java web applications. Specifically, we identify a set of *untrusted* data (i.e., data that can potentially carry malicious data that can be injected into the application)[1]. We also identify a set of *sensitive* data (i.e., data that manipulates the

[1]This data includes the values in fields of HTML forms, submitted URLs, HTTP requests, and content of cookies.

application)[2]. For each program, we examine whether there could be information flow from *untrusted* data to *sensitive* data (i.e. whether the application could be attacked by the injected malicious data). In our experiments, our analysis effectively detects a total of 26 real security vulnerabilities which do actually compromise the security of the applications.

The second client application is type inference. We perform experiments on the same set of Java web applications. Specifically, we identify an initial set of *untrusted* types — these are the variables identified as untrusted data in first client application, security violation detection. For each program, the analysis propagates the initial *untrusted* types and infers *untrusted* types for other variables in the program — these are the variables "tainted" by information flow from the initial *untrusted* set. In our experiments, the analysis effectively infers hundreds to thousands *untrusted* types, which could save significant effort in manually providing type annotations.

The last client application studies the effect of thread-shared variables on thread-local variables. We perform experiments on a set of multi-thread programs. Specifically, we identify a set of *thread-shared* variables and a set of *thread-local* variables. For each program, we study how many thread-local variables could be reached from thread-shared variables due to information flow (i.e. how many thread-local variables are affected by thread-shared variables). In our experiments, at least one third of the thread-local variables are affected by thread-shared variables.

We study the impact of implicit flow versus explicit flow on the three client applications. In security violation detection, implicit flow detects 14 additional real security violations compared to explicit flow. In type inference, implicit flow infers hundreds to thousands additional *untrusted* types. In the study of the effect of thread-shared variables, implicit flow detects 20% to 30% additional affected thread-local variables. Therefore, implicit flow has significant impact on these client applications.

[2]This data includes SQL execution commands, executed scripts, web page content, web cache, file access path, and shell commands.

1.2 Thesis Outline

The rest of this thesis is organized as follows. The general idea and problem setting of our framework is presented in Chapter 2, illustrated with a motivating example. Chapter 3 defines the run-time models that capture run-time *ownership*, *immutability* and *information flow*. Chapter 4 outlines the framework, and presents the ownership, immutability and information flow analyses. Chapter 5 discusses the experimental study. Chapter 6 describes the applications of the framework. Related work is discussed in Chapter 7. Chapter 8 presents a summary of the thesis and directions for future work.

CHAPTER 2
Overview And Motivating Example

The framework for inference of security properties can be summarized as: define run-time requirements as run-time models that capture these design level security properties; then infer via static analysis the security properties hold in the implementing code.

2.1 Motivating Example: POS System

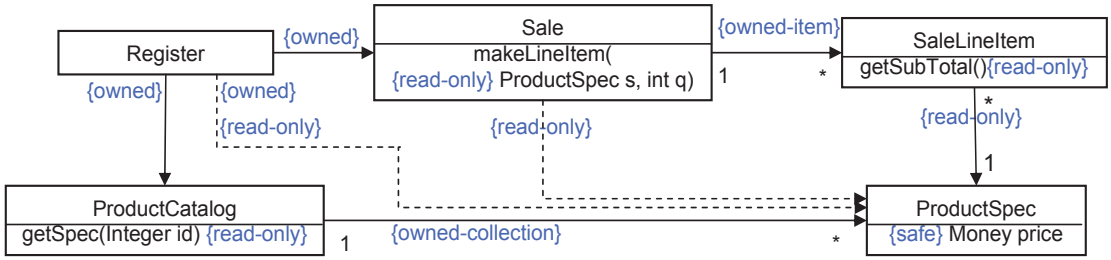

Figure 2.1: UML class diagram with security-related properties.

To illustrate the general idea of our framework, consider a component of a Point-of-Sale (POS) system [5], which includes the checkout procedure at the counter where transaction occurs. Figure 2.1 is the UML class diagram which shows the design of the Point-of-Sale system. The solid lines represent permanent associations (implemented through instance fields), and the dashed lines represent temporary dependencies (typically implemented through local variables). We have added constraints that illustrate the required security-related properties, which impose requirements on the implementation.

It is necessary to understand the POS system logic in order to clarify the security requirements of the POS system design. As in Figure 2.1, a `Register` object, the abstraction for the cash register, controls the sale logic. It creates a `ProductCatalog` object that stores the specifications of all products (i.e., the `ProductSpec` objects). The `Register` object creates a `Sale` object, initiates the sale, passes information

about sale items to the `Sale` object and completes the sale. When a new sale item is processed, the `Register` fetches the corresponding `ProductSpec` object from the catalog, and passes that object to the `Sale` object. The `Sale` object creates a new `SaleLineItem` object for each sale item and passes the `ProductSpec` object to it.

The clients of this component access public methods in `Register` to perform various tasks such as starting a new sale, entering a new sale line item, etc. These clients can be running on store terminals in the case of a classic brick-and-mortar POS system, or they can be running on remote computers in a the case of a web-based POS system.

Figure 2.1 presents a UML class diagram for the POS system which is annotated appropriately with required security-related properties: the *owned* constraints specify the property that the owned objects are part of the internal representation of the system and should not be leaked outside (i.e., to potentially untrusted clients). The *read-only* constraints specify the property that `ProductCatalog` is the "information expert" and the only object that can initialize and update product information. The *safe* constraint specifies the property that it should not be possible for a client to affect the price of a product (e.g., a malicious client could modify the product price or the computation of the sale total which could have disastrous consequences). Note that the ownership constraint does not prevent deeper information flow violations, such as leaks or modifications to sensitive data which is part of the owned objects.

2.2 POS System Security-related Properties Inferences

The security-related properties specified on UML diagrams impose requirements on the program run-time behaviors. It is necessary to define run-time models of the run-time ownership, immutability and information flow semantics. to capture the design level constraints. Then static analyses could be utilized to analyze the code according to the run-time models, and infer the implemented properties.

2.2.1 Run-time Models

The run-time models are defined to specify run-time semantics.

The ownership model is based on the notion of *owners-as-dominators* [2, 6, 3]. In Figure 2.1, the constraint that `Register` owns `ProductCatalog` requires that at runtime, the object $o_{Register}$ controls its field object $o_{ProductCatalog}$—the object $o_{Register}$ can create an object $o_{ProductCatalog}$, pass it to other parts of its representation, but cannot expose it to outside objects.

The immutability model is based on the modification of object state. In Figure 2.1, the constraint that the parameter `s` of type `ProductSpec s` in method `Sale.makeLineItem` is read-only requires that at runtime, the invocation of method $o_{Sale}.makeLineItem$ should not modify any part of its `s` parameter $o_{ProductSpec}$. The constraints that methods `ProductCatalog.getSpec` and `SaleLineItem.getSubtotal` are read-only forbid the run-time invocation of these methods from modifying any of their own parameters or any static fields. The constraint that `SaleLineItem` has read-only access to `ProductSpec` requires that any run-time method invocation on object $o_{SaleLineItem}$ should not modify any part of its field object $o_{ProductSpec}$.

The information flow model captures both explicit and implicit flow, and both shallow and deep flow. Intuitively, there is information flow from variable x to variable y, denoted by $x \mapsto y$, if changes in the input values of x are observable from the output values of y. In Figure 2.1, the `safe` constraint on field *price* (of type `Money`) in `ProductSpec` specifies that at runtime it should not be possible to have any information flow from any client of POS system to the object o_{Money} of the field *ProductSpec.price*.

2.2.2 Static Analyses

With the run-time models which illustrate run-time behavior requirements according to the design properties, the static analyses analyze the code, reason about its run-time behaviors and infer the implemented properties.

For completeness, we have included the Java code from [5, Chapter 20] in Appendix A.

When the ownership analysis is applied on the code in Appendix A, it concludes that $o_{Register}$ is the immediate dominator of $o_{ProductCatalog}$ in the summary object graph, which shows access relationships between run-time objects over all

possible execution paths. According to the run-time ownership model, the analysis reverse engineered an `owned` constraint on the association from `Register` to `ProductCatalog`. Thus, the implementation in Appendix A follows the constraint that `Register` owns `ProductCatalog` in Figure 2.1.

The immutability analysis of the code in Appendix A concludes that every possible invocation of $o_{Sale}.makeLineItem$ does not modify the transitive object state of the parameter `s`, and every possible invocation of method `getSpec` in class `ProductCatalog` does not modify any object state observable from outside of method calls. Thus the analysis reverse engineered `read-only` constraints on the parameter `ProductSpec s` in method `Sale.makeLineItem`, and on the method `ProductCatalog.getSpec`.

Surprisingly, the immutability analysis reports a modification of the field $SaleLineItem.ProductSpec$ by method invocation of `SaleLineItem.getSubtotal`. Thus method `SaleLineItem.getSubtotal` and the association between `SaleLineItem` and `ProductSpec` were reported as non-`read-only`. A brief examination of the code revealed a bug that could be very serious—the `SaleLineItem` object mistakenly modified the *price* field of the `ProductSpec` object, due to the fact that the times method changes the value of the receiver object (line 25 in Appendix A, see also Section 4.4.1). As a result, subsequent sales fetched `ProductSpec` with wrong prices and computed incorrect sale totals!

Subsequently, the information flow analysis reports that a malicious party could select multiple items of the same product, to modify the item price on purpose, resulting in information flow from the malicious party to field *ProductSpec.price*. Thus the analysis concludes that field *ProductSpec.price* is `un-safe`.

2.3 Framework Setting

The framework works not only on complete programs but also on incomplete programs (i.e., software components) as well. This is an important feature because reasoning about security should be performed on software components; any realistic program understanding and verification tool should be able to work on both software components and complete programs.

Therefore, the problem statement should consider the setting of both complete programs and software components. Particularly, here we define the working problem of ownership, immutability and information flow constraints.

2.3.1 Analysis of Whole Programs

Conventionally, software tools infer security-related properties by examining field associations or elements such as methods and parameters.

The *ownership property* and *immutability property* are inferred on instance fields and variables of reference types in the code.[3] The ownership inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$ (field f of object o points to o'), o owns o'. Similarly, the immutability inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, no method invocation on o mutates o'.

The *immutability property* is inferred on method parameters, and on methods. A parameter p in method m is inferred as `read-only` if for all run-time objects o_{p_i}, invocations of m do not mutate the heap structure rooted at the object referred by p. A method m is inferred as `read-only` if no run-time invocation of $o.m$ modifies the visible state: parameters of m, and all static fields that are initialized before method invocation.

The *information flow property* is inferred for sensitive fields and variables. A field or variable is inferred as `confidential` if there is no information flow from that field or variable to an untrusted part of the code (i.e., a *sink*). A field or variable is inferred as `safe` if there is no information flow from an untrusted part of the code (i.e., a *source*) to that field or variable.[4]

2.3.2 Analysis of Software Components

We consider the following problem setting for software components. Let *Cls* denote a Java component—that is, a set of interacting Java classes which includes all

[3]The rest of the thesis focuses on permanent associations (implemented with instance fields). Although our framework and the augmented analyses are general and can handle temporary dependencies, we omit their discussion for clarity.

[4]Users can choose to identify the *sinks* and the *sources*. Without users' specification, the *sinks* are defined as outputs from the programs, and the *sources* are defined as inputs into the programs.

transitively referenced classes. A subset of these classes are designated as *accessible* and client code can access the component through *accessible* fields and *accessible* methods in these classes. we define the classes in *Cls* as trusted while the client code built on top of these classes is defined as untrusted.

Consider the code in Appendix A which implements the POS system design from Figure 2.1; it is taken from [5, Chapter 20] with minor modifications introduced for brevity. Classes `Register`, `Sale`, `ProductCatalog`, `SaleLineItem` and `ProductSpec` are in *Cls*. Class `Register` and its methods are accessible while the rest of the classes have package visibility and are not directly accessible from client code.

We employ the following constraint, which is standard for other problem definitions that require analysis of incomplete programs [7, 8, 9]. We only consider executions in which the invocation of a method in component does not leave *Cls*—that is, all of its transitive callees are also in *Cls*. If we consider the possibility of unknown subclasses, all instance calls from *Cls* could be "redirected" to unknown external code that may affect the information flow inference. For example, a field may be confidential in the current set of classes but an unknown subclass may override a method and leak the field (e.g., by using the confidential field in a computation of the value of a public static field).

Thus, *Cls* is augmented to include the classes that provide component functionality as well as all other classes transitively referenced. In the experiments presented in Chapter 5 we included all classes that were transitively referenced by *Cls*. This approach restricts analysis information to the currently "known world"—that is, the information may be invalidated in the future when new subclasses are added to *Cls*. One could change the analysis to make worst case assumptions for calls that may enter unknown methods; however, in that case the analysis would be overly conservative and would unlikely report useful information.

CHAPTER 3
Run-time Models

To infer security properties illustrated as by constraints, we need run-time models to define runtime semantics of ownership, immutability, information flow etc. We present run-time models that capture the meanings of ownership, immutability, confidentiality and integrity at design level, by identifying useful run-time object access and information flows. These models help to understand these security properties and instruct how to identify violations against the security properties in program execution.

Below, we briefly describe the models for the ownership, immutability and information flow constraints. We envision that the framework can be augmented with additional useful constraints and corresponding static analyses.

3.1 Notations

Throughout the thesis we use the following notational convention.

Run-time objects are denoted using superscript r: e.g., o^r, o_i^r, o_1^r, etc.

Analysis objects (i.e., abstract objects that represent the run-time objects) are denoted without the superscript r: e.g., o, o_i, o_1, etc; our analysis represents objects by their allocation site: for example, all run-time objects allocated at allocation site s_i are represented by analysis object o_i.

Local variable v in method m is denoted as `m.v`. Field f referenced through local variable v in method m is denoted as `m.v.f`.

The `this` in method m is denoted as $\texttt{this}_m$.

3.2 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [2, 6, 3]. In this model, each execution point is represented by an *object graph* which shows access relationships between run-time objects. There is an edge $o^r \rightarrow o_1^r$ from run-time object o^r to run-time object o_1^r (i.e., we say that o_1^r is accessed in the *context*

```
public class Vector {
     protected Object[] data;
     public Vector(int size) {
1         data = new Object[size]; }
     public void add(Object e,int at) {
2         data[at] = e; }
     public Object elementAt(int at) {
3         return data[at]; }
     public Iterator iterator() {
4         return new VIterator(this); }
}

final class VIterator implements Iterator {
     Vector vector;
     int count;
     VIterator(Vector v) {
5         this.vector = v;
6         this.count = 0; }
     Object next() {
7         Object[] data = vector.data;
8         int i = this.count;
9         this.count++;
10        return data[i]; }
}

main() {
11   Vector v = new Vector(100);
12   X x = new X();
13   v.add(x,0);
14   Iterator i = v.iterator();
15   x = (X) i.next();
16   x.m();
}
```

Figure 3.1: Simplified vector and its iterator.

of o^r) if and only if one of the following is true:

- Reference instance field f in o^r refers to o_1^r.
- Object o^r is an array object with element o_1^r.
- An instance method or constructor invoked on receiver o^r has local variable r that refers to o_1^r, or a static method called from an instance method or constructor invoked on o^r has a local variable r that refers to o_1^r.[5]

Note that the first two items account for somewhat permanent, "heap" accesses. In contrast, the last item accounts for temporary, "stack" accesses that appear when a local is set to point to an object, and disappear when the method enclosing the local finishes execution.

In accordance with the owners-as-dominators model, we say that o^r *owns* o_1^r if and only if o^r is the immediate dominator of o_1^r in the object graph at all time points of execution. Node m *dominates* node n if every path from the root of the graph that reaches node n passes through node m. The root dominates all nodes. Node m *immediately dominates* node n if m dominates n and there is no node p such that m dominates p and p dominates n.

Consider the partial object graph in Figure 3.2; it is an object graph created during the execution of `main` in Figure 3.1. Node o_{root}^r represents the start of program execution. The other nodes correspond to the runtime objects created at the appropriate allocation sites in Figure 3.1. o_{Vector}^r does not own $o_{Object[]}^r$ because during the execution of `next` there is a temporary access from o_{VIter}^r to $o_{Object[]}^r$. o_{Vector}^r would own $o_{Object[]}^r$ if `next` were never executed (i.e., line 15 were removed from `main`).

Essentially, this model requires that an owner object controls the owned object—the owner can create an owned object, pass it to other parts of its representation, but cannot expose it to outside objects. This model intuitively captures the notion of ownership and composition in modeling [2].

[5]We require that there be an explicit reference variable for each object that is accessed (i.e., a statement $r.m().n()$ is re-written into an equivalent sequence r_1=$r.m()$; $r_1.n()$;).
We require that all newly created objects appear in the object graph explicitly [2]. That is, at the point of creation a new object is stored in a new local variable(i.e., a statement $r.m(new\ A())$ is re-written into an equivalent sequence r_1=$new\ A()$; $r.m(r1)$;).

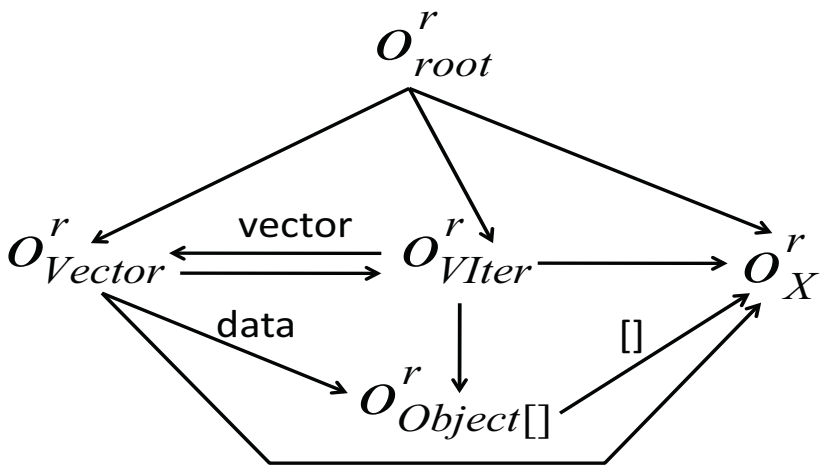

Figure 3.2: Runtime Object graph for Figure 3.1.

The ownership inference problem for complete programs therefore is to find the fields f such that for each run-time edge $o^r \xrightarrow{f} o^r_1$, o^r owns o^r_1 according to the above definition. The ownership inference problem for components therefore is to find the fields f such that for all possible client code, for each run-time edge $o^r \xrightarrow{f} o^r_1$ (field f of o^r points to o^r_1), o^r owns o^r_1 according to the above definition. The associations through these fields are marked as `owned`.

3.3 Immutability Model

Let e be an execution of a method m on receiver object o^r; e modifies an object o^r_1 if it triggers a change in the object structure rooted at o^r_1—that is, e leads to a statement $p.f = q$ which writes some o^r_2 reachable from o^r_1 (i.e., p refers to o^r_2 and there is a path of field edges from o^r_1 to o^r_2). For example, the execution of `add` with receiver o^r_{Vector} (line 13 in Figure 3.1) modifies $o^r_{Object[]}$.

A parameter p_i of method $m(...p_i...)$ is `read-only` if no execution of method m modifies the object o^r referenced by p_i. Consider the execution of method `add` with run-time receiver o^r_{Vector} from Figure 3.1. The field `data`, which is partial object state of o^r_{Vector}, is reset by the method execution. Thus the *this* parameter of method `add`, the receiver object o^r_{Vector}, is modified by the method execution.

A method m is `read-only` if the following two conditions are true: 1) all

parameters of m are `read-only`, and 2) no execution of m modifies an object referenced by a static field. As in Figure 3.1, method `add` is not read-only, as its *this* parameter is modified by its execution.[6]

Finally, we say that o^r *has read-only access to* o_1^r if no execution of a method on receiver o^r modifies o_1^r. An association from class A to class B is `read-only` if and only if for every execution of the code, each instance of A has read-only access to the corresponding instances of B. Consider run-time edge $o^r_{Vector} \stackrel{data}{\rightarrow} o^r_{Object[]}$ from Figures 3.2 and 3.1. Clearly, o^r_{Vector} does not have read-only access to array $o^r_{Object[]}$ because the execution of method `add` with receiver o^r_{Vector} (line 13) modifies $o^r_{Object[]}$. Note that the model considers *transitive* modifications (i.e., an execution may modify fields of o_1^r as well as fields of fields of o_1^r, and so on.). As another example, consider edge $o^r_{VIter} \stackrel{vector}{\rightarrow} o^r_{Vector}$. o^r_{VIter} has read-only access to o^r_{Vector} because the execution of `next` with receiver o^r_{VIter} (line 15) does not modify the structure rooted at o^r_{Vector}.

The model does not treat constructor invocations and the corresponding initialization statements `this.f=q` as modifications of the newly constructed object. This is done in order to capture better the intuitive meaning of immutability in the context of class diagrams.

The immutability inference problem for complete programs therefore is to find the fields f such that for each run-time edge $o^r \stackrel{f}{\rightarrow} o_1^r$, o^r has read-only access to o_1^r. The immutability inference problem for components therefore is to find the fields f such that for all possible client code, for each run-time edge $o^r \stackrel{f}{\rightarrow} o_1^r$, o^r has read-only access to o_1^r. The associations through these fields are marked as `read-only`.

3.4 Information Flow Model

Intuitively, there is information flow from variable x into variable y, denoted by $x \mapsto y$, if changes in the values of x are observable from the values of y. Such flows are *direct* and *indirect* [4, 10]. Direct flows can be *explicit* (i.e., data-flow based) and *im-*

[6]Note that this definition of immutable method misses returned objects (i.e., it does not include the entire visible state). That is, a method m can create an object and return this object to the caller, but our definition would still consider m immutable. Although we believe that our choice is appropriate, we acknowledge that it is not the standard choice. However, the choice is arbitrary—the definition and corresponding analyses can be trivially changed to accommodate other choices.

plicit (i.e., control-flow based). Direct explicit flows arise at assignment statements: for example, for statement `x=y+5` there is direct explicit flow $y \mapsto x$. Direct implicit flows arise from conditionals: for example, for statement `if (x>1) then y = w;` there is direct implicit flow $x \mapsto y$ since changes of the values of x are observable from the values of y. Indirect (i.e., transitive) flows arise from compositions of direct flows: for example, for the sequence of statements `y=z+w; x=y+5;` there are direct flows $z \mapsto y, w \mapsto y, y \mapsto x$ which lead to indirect flows $z \mapsto x$ and $w \mapsto x$.

We formalize the intuitive flow semantics described above. We consider each Java statement kind and the direct flows that arise from it at runtime:

- **Assignment** $l = (...operator)\ r$ leads to explicit flow $r \mapsto l$, and implicit flows $v_i \mapsto l$ such that this assignment is in scope of some conditional statement(*if, case, while, do while*) with v_i as conditional variable.
  For example, consider `if (x>1) then y = w;`. There is implicit flow $x \mapsto y$, since the statement `y = w;` is in scope of the *if* statement where x is the conditional variable.

- **Instance field write** $l.f = r$ leads to explicit flow $r \mapsto o^r.f$, where o^r is the run-time object referenced by l at the point of execution of the statement, and implicit flows $v_i \mapsto o^r.f$ s.t. this write is in scope of some conditional statement(*if, case, while, do while*) with v_i as conditional variable,

- **Instance field read** $l = r.f$ leads to explicit flow $o^r.f \mapsto l$, and implicit flows $v_i \mapsto l$ s.t. this read is in scope of some conditional statement(*if, case, while, do while*) with v_i as conditional variable. Again, o^r is the run-time object referred to by r at the point of execution of the statement.

- **Method call** $l = r_0.m(r_1, ...)$ dispatched to method $m'(this, p_1, ..., ret)$ leads to flows $r_0 \mapsto this$, $r_1 \mapsto p_1$, ... and $ret \mapsto l$, and implicit flows $v_i \mapsto m'$, $v_i \mapsto this$, $v_i \mapsto p_1$, ... $v_i \mapsto l$, s.t. this method call is in scope of some conditional statement(*if, case, while, do while*) with v_i as conditional variable. Here *this* denotes the implicit parameter `this` of method m', the p_i denote the formal parameters of m', and *ret* denotes a special variable that holds the return value of m'.

We distinguish two types of indirect flow. There is *shallow flow* from variable l into variable r if there is a sequence of statements, executed in order, that leads to indirect flow from l to r. For example, suppose l_2 and l_3 both point to object o^r, the execution of statement $l_1 = l + x$ leads to flow $l \mapsto l_1$, then the execution of $l_2.f = l_1$ leads to flow $l_1 \mapsto o^r.f$, and then the execution of $l_4 = l_3.f$ leads to flow $o^r.f \mapsto l_4$. Finally the execution of $r = l_4 - y$ leads to flow $l_4 \mapsto r$.

Note that when l is a reference variable, there may be flow from the object structure rooted at l. There is *deep flow* from l into r if there is shallow flow from some l' into r, where l' is an alias of $l.f_1.f_2...f_k$ (i.e., l' points to an object o_1^r which can be reached on a sequence of field dereferences from the object o^r referred to by l).

The confidentiality problem for complete programs is: given a variable s and the above definition of information flow, does there exist any information flow from s into some untrusted variable v (such as the parameters of `write` methods of `OutputStreams`)? The dual integrity problem for complete programs is: given a variable s and the above definition of information flow, does there exist any information flow from some untrusted variable v (such as the parameters or return value of `read` methods of `InputStreams`) to variable s?

For components, the confidentiality problem is the following: given a variable s in component *Cls* and the above definition of information flow, does there exist a client that would permit information flow from s into some variable v in the client code? Analogously, the integrity problem for components is: given a variable s in component *Cls* and the above definition of information flow, does there exist a client that would permit information flow from some variable v in the client code into s?

Example. Consider package `zip` in Figure 3.3. This example, adapted from one of our benchmarks, is based on the classes from the standard library package `java.util.zip`; some modifications are made to better illustrate the problem and the analysis. Classes `ZipInputStream` and `ZipEntry` are public and therefore accessible; interface `ZipConstants` has package visibility and therefore it is not directly accessible. All public methods and fields in `ZipInputStream` and `ZipEntry` are accessible.

```
package zip;
public class ZipInputStream {
    private ZipEntry entry;
1   private byte[] tmpbuf = new byte[512];
    public ZipEntry getNextEntry() {
2       ZipEntry e = readLOC();
3       this.entry = e;
4       return this.entry;
    }
    private ZipEntry readLOC() {
5       ZipEntry e = new ZipEntry();
6       long i1 = get32(tmpbuf,LOCFLG);
7       e.flag = i1;
8       long i2 = get32(tmpbuf,LOCSIZ);
9       e.size = i2;
10      return e;
    }
    private static int get16(byte b[], int off) {
11      byte b1 = b[off];
12      int i1 = b1 & off;
13      return i1;
    }
    private static long get32(byte b[], int off) {
14      long i1 = (long) get16(b,off);
15      long i2 = (long) get16(b,off+2);
16      int i3 = i1 | i2;
17      return i3;
    }
} // end of class ZipInputStream

public class ZipEntry {
    long flag;
    long size = -1;
    public void setSize(long size) {
18      this.size = size;
    }
    public long getSize() {
19      return this.size;
    }
}
interface ZipConstants {
    static final long LOCFLG = 6;
    static final long LOCSIZ = 18;
}
```

Figure 3.3: Sample package `zip`.

First, consider local variable `e` in method `readLOC`. It is easy to see that one can write a client of this component which exposes shallow flow from `e` to the client as following:

```
ph_ZIS = new ZipInputStream();
ph_ZE = ph_ZIS.getNextEntry();
```

Let o^r_{ZIS} stand for the run-time `ZipInputStream` object created in the above client. This client exhibits these shallow follows: `readLOC.e` $\mapsto$ `readLOC.ret` due to line 10, `readLOC.ret` $\mapsto$ `getNextEntry.e` due to line 2, `getNextEntry.e` $\mapsto$ o^r_{ZIS}`.entry` due to line 3, o^r_{ZIS}`.entry`$\mapsto$ `getNextEntry.ret` due to line 4, and `getNextEntry.ret` $\mapsto$ `ph_ZE` due to the call to `getNextEntry` in the client. These shallow flows result in the shallow flow from `e` to `ph_ZE` in the client.

Second, consider flow from field `tmpbuf` in `ZipInputStream`. Let o^r_{ZE} stand for the run-time `ZipEntry` object created at line 5. One cannot write a client which exposes shallow flow from `tmpbuf` (i.e., the reference to the array `tmpbuf` is never exposed to a client). However, one can write a client which exposes deep flow from `tmpbuf` (i.e., the content of the array is exposed). Consider client

```
ph_ZIS = new ZipInputStream();
ph_ZE = ph_ZIS.getNextEntry();
ph_long = ph_ZE.getSize();.
```

The aliasing of `tmpbuf` and `get16.b` is triggered by the invocation of `getNextEntry`, which invokes `readLOC` at line 2, then `get32` at line 8, and `get16` at lines 14-15. Thus the dereference `get16.b[off]` at line 11 is an alias of `tmpbuf[]`. Therefore, the shallow flow `get16.b[off]` $\mapsto$ `get16.b1` $\mapsto$ `get16.i1` $\mapsto$ `get16.ret` $\mapsto$ `get32.ret` $\mapsto$ `readLOC.i2` $\mapsto$ o^r_{ZE}`.size` $\mapsto$ `getSize.ret` $\mapsto$ `ph_long`, results in a deep flow from `tmpbuf` to `ph_long` in the client.

CHAPTER 4
Static Analysis Framework

This section outlines our static analysis framework. Clearly, the problems outlined in the previous section require analysis of partial programs (i.e., components) as well as complete programs. We address the issue of dealing with components by employing a general technique called *fragment analysis* [11, 7] which enables analysis of partial programs (Section 4.1). Furthermore, the problems require points-to information and we employ a general-purpose context-insensitive points-to analysis (Section 4.2).

The fragment analysis and the points-to analysis are a general foundation that allows building of different client analyses. The context-sensitivity is recovered in these client analyses. So far we have built analyses that infer ownership, immutability and information flow in accordance with the models outlined in the previous chapter. They work directly on Java code and do not require annotations by the programmer; also, they work on both complete programs and on software components. The analyses infer constraints in accordance with these models. We envision that the framework will be augmented with other constraints of interest and corresponding client analyses.

4.1 Fragment Analysis

Clearly, the problem considered in this paper requires analysis of incomplete programs. The input is a set of classes *Cls* and the analysis needs to approximate information flow that is valid across all possible executions of arbitrary client code built on top of *Cls*. To address this problem we make use of a general technique called *fragment analysis* due to Nasko Rountev [11, 7]. Fragment analysis works on a program fragment rather than on a complete program; in our case the fragment is the set of classes *Cls*.

Initially, the fragment analysis produces an artificial `main` method that serves as a placeholder for client code written on top of *Cls*. Intuitively, the artificial `main` simulates the possible flow between *Cls* and the client code. Subsequently, the

fragment analysis attaches `main` to *Cls* and uses whole-program analysis to compute information that approximates flow over all possible clients of *Cls* [11, 7].

The artificial `main` method contains a variety of placeholder statements. For each class $X \in Cls$, there is a placeholder variable ph_X that serves as a representative for all unknown external reference variables of type X (i.e., all such variables that may occur in some client code). There are also placeholder statements that represent the effect of accessing fields and methods in *Cls*. Finally, there are placeholder statements represent the possible effects of assigning variables of one type to variables of another type (including the effects of possible casting).

The placeholder `main` method for our running example is given at the end of Appendix A. The method contains placeholder variables for types from *Cls* that can be accessed by client code. It also contains statements that represent all possible interactions involving *Cls*; their order is irrelevant because our analyses are flow-insensitive. Generally, `main` invokes all public methods from the classes in *Cls* designated as accessible. For details on the fragment analysis see [11, 7].

4.2 Points-to Analysis

Points-to analysis is a well-known program analysis. It finds the objects that a given reference variable or a reference object field may point to. Points-to information is needed by all of our client analyses; most likely it will be needed by future client analyses as well. There is a wide variety of points-to analyses, with different degrees of precision and cost. Our work uses the known and relatively well-understood and precise Andersen's points-to analysis [12, 13]. We choose this context-sensitive analysis with context-sensitivity recovered in the client analyses, rather than a context-sensitive analysis, for a tradeoff between precision and efficiency. This analysis is flow-insensitive, context-insensitive and inclusion-based; it uses an analysis variable for each reference variable, and an object name for each allocation site.

The points-to analysis is defined in terms of three sets. Set R is the set of locals, formals and static fields of reference type. Set O is the set of object names o_i; each $o_i \in O$ represents all the objects created at an allocation site s_i. Set F

contains all instance fields in program classes. The analysis solution is a *points-to graph Pt* where the edges represent the following "may-refer-to" relationships.

- Let $r \in R$ and $o \in O$. An edge $(r, o) \in Pt$ means that at runtime r may refer to some object that is represented by o.
- Let $f \in F$ be a reference instance field in objects represented by some $o \in O$. An edge $(o.f, o_2) \in Pt$ means that at run time field f of some object represented by o may refer to some object represented by o_2.
- Let o represent the array objects. An edge $(o[\ ], o_2) \in Pt$ shows that some element of some array represented by o may refer at run time to an object represented by o_2.

Most points-to analyses, including Andersen's, are formulated as whole-program analyses. The placeholder `main` method constructed by the fragment analysis "completes" the component and thus enables the use of whole-program points-to analysis on the completed component. The `main` method simulates all possible clients that could be built on top of *Cls* and thus the result of the whole-program points-to analysis includes all points-to graphs that could result from individual clients [11, 7].

4.3 Ownership Analysis

The output of the points-to analysis is needed to construct the *approximate object graph Og* which approximates all possible run-time object graphs. Subsequently, *Og* is used for ownership inference.

4.3.1 Approximate Object Graph

The nodes in *Og* are taken from the set of object names O and the edges represent "may-access" relationships. Figure 4.1 outlines the construction of *Og* given a points-to graph *Pt* described in Section 4.2. The analysis starts with an empty *Og* and adds edges to *Og* as it processes program statements. Intuitively, the algorithm tracks flow of objects from one context to another context. There are five kinds of edges: (i) create edges (due to object creation), (ii) in edges (due to

procedure **objectGraph**
input *Stmt*: set of statements *Pt*: points-to graph
output $Og : O \rightarrow \mathcal{P}(O)$
[1] **foreach** statement s in method m
$s_i : l = new\ C(...)$
[2] add $\{c \rightarrow o_i \,|\, c \in \mathcal{C}_m\}$ to Og
// **creation** edge
[3] **foreach** statement s in method m
$s : l = r.f$ s.t. $r \neq$ `this`
$s : l = r.n(...)$ s.t. $r \neq$ `this`,
[4] add $\{c \rightarrow o_j \,|\, c \in \mathcal{C}_m \wedge (l, o_j) \in Pt\}$ to Og
//**out** edge: from callee contexts to caller contexts
[5] **foreach** statement s in method m
$s : l.f = r$ s.t. $l \neq$ `this`
$s : l.n(r)$ s.t. $l \neq$ `this`,
$s : l = new\ C(r)$,
[6] add $\{o_i \rightarrow o_j \,|\, (l, o_i) \in Pt \wedge (r, o_j) \in Pt\}$ to Og
//**in** edge: into callee contexts from caller contexts
[7] **foreach** statement s in method m
$s : l = r.m($`this`$)$,
$s : l.f =$ `this`,
$s : l =$ `this`,
[8] add $\{o_i \rightarrow o_i \,|\, (\texttt{this}_m, o_i) \in Pt\ \}$ to Og
//**self** edge: into callee contexts from caller contexts
[9] label with f each $o_i \rightarrow o_j \in Og$ s.t. $(o_i.f, o_j) \in Pt$

$$\mathcal{C}(m) = \begin{cases} \{c \,|\, (\texttt{this}_m, c) \in Pt\} & \text{if } m \text{ is an instance method} \\ \bigcup_{m' \in Callers(m)} \mathcal{C}(m') & \text{if } m \text{ is a static method} \end{cases}$$

Figure 4.1: Construction of Approximate Object Graph Og. $\mathcal{P}(X)$ denotes the power set of X. Og is initially empty.

arguments), (iii) out edges (due to return), (iv) self edges (due to leak of this), and (v) field edges. An edge $o_i \rightarrow o_j$ may be of more than one kind. The significance of these kinds of edges will become clear in Section 4.3.2.

Lines 1-2 account for edges due to object creation; they are edges from each context of method m to c, to the object created at that site. These edges are labeled as create edges. They capture run-time object creation: when an object is created, this newly created object becomes available in the context of the caller.

The contexts of the caller m are stored in set $\mathcal{C}_m$. If m is an instance method, $\mathcal{C}_m$ equals to the points-to set of the implicit parameter `this` of m. If m is a static method, $\mathcal{C}_m$ includes the points-to sets of all implicit parameters `this` of instance methods n reachable backwards from m on a chain of static method calls (i.e., $\mathcal{C}_m$ includes the closest instance contexts enclosing m); if `main` is reachable backwards from m on a chain of static method calls, $\mathcal{C}_m$ includes the special context `root` as well. At allocation sites new edges are added to Og from each context of the enclosing method m to the object name that represents the newly created object.

Lines 3-4 account for edges due to flow out from other contexts to the context of m. An instance field read $l = r.f$, $r \neq$ `this`, and a virtual call $l = r.n(r_1)$, $r \neq$ `this` in method m, result in edges from every context c of m to each o_j in the points-to set of l. These edges are labeled as *out edges*. They capture run-time access "due to return": when an object o_j, accessible to the object referred by r, is "returned" due to this statement, it becomes accessible to the context of m.

Lines 5-6 account for edges due to flow from the contexts of m into other contexts. An instance field write $l.f = r$, $l \neq$ `this`, a virtual call $l.n(r)$, and an object creation statement $s : l = new\ C(r)$ in method m, result in edges from every object o_i in the points-to set of l to every object o_j in the points-to set of r. These edges are *in edges*. They capture access "due to arguments": when an object referred by r (and accessible to the context of m) is assigned to $l.f$ or passed to l , this object becomes accessible to the object that l refers to.

Lines 7-8 account for edges due to statements where implicit parameter `this` is leaked (namely, statements $l = r.m($`this`$)$, $l.f =$ `this`, and $l =$ `this`). These edges are self edges. These edges account that the `this` object accesses itself, and thus, it may pass a reference to itself to other objects.

Finally, line 9 examines each edge $o_i \rightarrow o_j$ in the constructed Og, and if o_j is in the points-to set of some field f of o_i (i.e., $(o_i.f, o_j) \in Pt$), the edge is determined to be a field edge and is labeled with $f : o_i \xrightarrow{f} o_j$.

As an example, consider the code in Figure 3.1. In this case, the `objectGraph` algorithm in Figure 4.1 constructs precisely the approximate object graph in Fig-

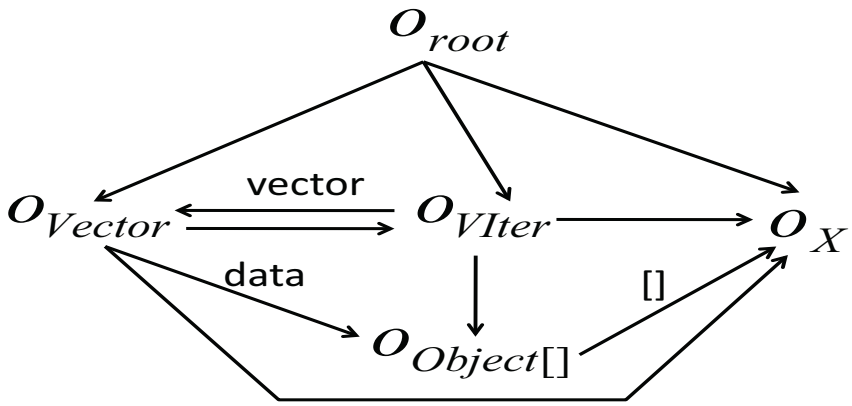

Figure 4.2: Approximate Object graph for Figure 3.1.

ure 4.2.[7] Edges $o_{root} \rightarrow o_{Vector}$, $o_{root} \rightarrow o_X$ and $o_{Vector} \rightarrow o_{Object[]}$ are due to creations at code lines 11, 12 and 1 respectively (captured by lines 1-2 in the algorithm). Edge $o_{Vector} \rightarrow o_X$ is due to parameter passing at code line 13 (captured by lines 5-6 in the algorithm). Edge $o_{Object[]} \rightarrow o_X$ is due to element storing at code line 2 (captured by line 9 in the algorithm). Furthermore, edge $o_{root} \rightarrow o_{VIter}$ is due to code line 14, and edges $o_{Vector} \rightarrow o_{VIter}$ and $o_{VIter} \rightarrow o_{Vector}$ are due to line 4. Finally, edges $o_{VIter} \rightarrow o_{Object[]}$ and $o_{VIter} \rightarrow o_X$ are due respectively to lines 7 and 10 in `next`.

Note that the analysis ignores statements through `this` (`this`.$f = r$, $l =$ `this`.f and `this`.$n(r)$) and direct assignments (l = r); this is correct because these statements do not result in new run-time edges. For example, consider `this`.$f = r$ in method m; the run-time access edge between the receiver of m and the object referred to by r is already there; it is due to object creation (e.g, $r = new\ C()$), due to a return (e.g., $r = r_1.n()$), or it is due to arguments.

A naive analysis (one that considers statements through `this`) will inherit imprecision from the context-insensitive Andersen's points-to analysis.[8] For example, consider the following common object-oriented code, which initializes an instance field through an instance method. Classes `Y` and `Z` inherit from `X`.

[7] In this example, the approximate object graph is the same as the run-time object graph and thus is very precise.

[8] We choose relatively precise Andersen's points-to analysis rather than a context-sensitive points-to analysis for the benefit of efficiency.

```
class A {
   X f;
   m(X xa) { this.f = xa; }
}
A a1 = new A(); // o_a1    a1.m(new Y()); // o_y
A a2 = new A(); // o_a2    a2.m(new Z()); // o_z
```

The context-insensitive points-to analysis merges the contexts of invocation of method `m`, which results in having fields f of each of the `A` objects, o_{a1} and o_{a2}, point to both o_y and o_z. Using the naive analysis which considers statement `this.f=xa` in method `m`, will result in access edges from o_{a1} to both o_y and o_z, and from o_{a2} to both o_y and o_z. This is imprecise — o_{a1} accesses only o_y, and o_{a2} accesses only o_z.

The object graph analysis in Figure 4.1 handles this case and other idiomatic cases precisely. It ignores `this.f=xa`, the statement that would have caused imprecision; it processes statements `a1.m(new Y())` and `a2.m(new Z())`; the first statement results in an access edge from o_{a1} to o_y, and the second statement results in an edge from o_{a2} to o_z.

4.3.2 Ownership Inference

As described in Section 3.2, owners-as-dominators defines ownership on the dominance relation in the run-time object graph. Thus the static ownership analysis needs to infer dominance on the approximate object graph Og constructed in Section 4.3.1.

However, the ownership inference could not be easily done by using known dominator algorithms on the approximate object graph. Because the object graph Og is a *static* representation of objects and object accesses — that is, a node in the object graph typically correspond to multiple run-time objects and an edge corresponds to multiple run-time access edges. Using standard dominator algorithms on the object graph would affect both correctness and precision. For example, consider the object graph in Figure 4.3 for code in Figure 4.4. A dominator algorithm will determine that `Y`'s container, o_{cy}, does not dominate its array, o_d, because of the multiple paths from `root` to o_d that do not go through o_{cy}. However, this is

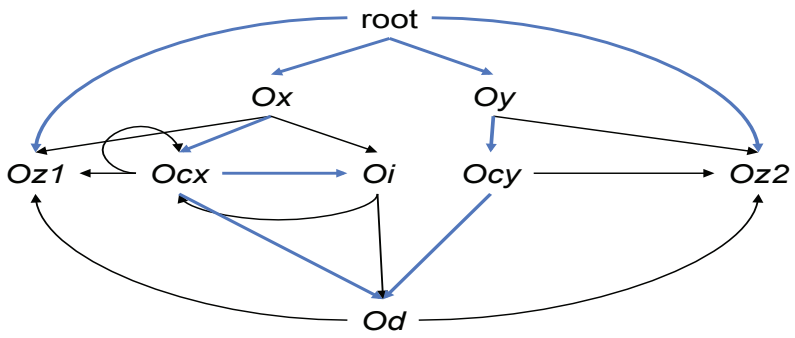

Figure 4.3: Object graph for Code in Figure 4.4.

```
class Main {
  public static void main(String[] args) {
1   X x = new X();              //ox
2   Z z1 = new Z(); x.mx(z1);//oz1
3   Y y = new Y();              //oy
4   Z z2 = new Z(); y.my(z2);//oz2
  }
}
class X {
  Container cx;
  void mx(Z zx) {
5   cx = new Container(10);   //ocx
6   cx.put(zx,0);
7   Iterator itx = cx.getIt();
  }
}
class Y {
  Container cy;
  void my(Z zy) {
8   cy = new Container(10);   //ocy
9   cy.put(zy,0);
  }
}
```

```
class Container {
  Object[] data;
  Container(int size) {
10  data = new Object[size];  //od
  }
  void put(Object o, index i) {
11  data[i] = o;
  }
  Iterator getIt() {
12  return new Iterator(this);//oi
  }
}
class Iterator {
  Object[] data;
  Iterator(Container c) {
13  data = c.data;
  }
}
```

Figure 4.4: Example illustrating imprecision of dominator algorithm.

imprecise as at runtime, o_{cy} and o_{cx} point to different array and do dominate their own arrays.

Our ownership inference uses the object graph Og to reason about object ownership. Consider the partial object graph in Figure 4.5, extracted from the code in Appendix A from [5]. Node `root` represents the special context of `main` and node o_r represents the `Register` object (created in `main`). o_{ps1} represents `ProductSpec` objects (created in `ProductCatalog`), o_m represents `Money` objects

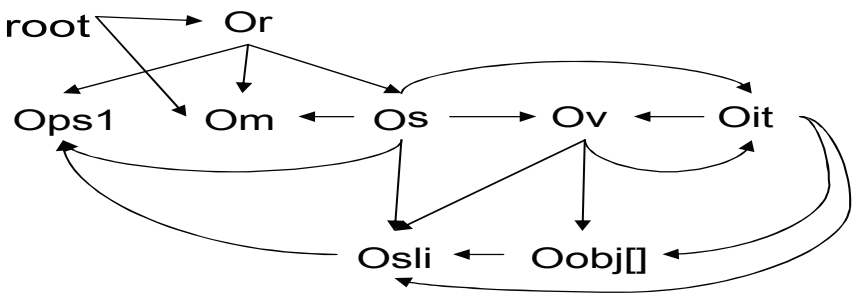

Figure 4.5: Partial *Og* for Appendix A.

(created in `main` to account for payment for a sale), and o_s represents `Sale` objects (created in `Register` when initiating a new sale). o_{sli} represents `SaleLineItem` objects (created in `Sale` when processing a new line item) and o_v represents the collection needed to store the `SaleLineItem`s. Finally, o_{it} represents iterators over the collection of `SaleLineItem`s (used in `Sale` when calculating the sale total), and $o_{obj[]}$ represents array that holds the contents of the collection.

To reason about ownership on an edge $o_i \rightarrow o_j$ in the object graph, the inference analysis computes a dominance boundary of object o_i, which is a subgraph of *Og* rooted at o_i such that all objects in the subgraph are owned by o_i. The computation of the dominance boundary is based on the following observations.

***Create* Reachability** One important observation is that in order for an object o_j to be in the boundary of o_i, o_j *must have been created* by o_i, directly or indirectly. Let set *createClosure*(o_i) include o_i and the set of objects reachable on *create* edges from o_i in the object graph *Og*. In the running example from Appendix A and the partial object graph in Figure 4.5, *createClosure*(o_r) = $\{o_r, o_{ps1}, o_s, o_{sli}, o_v, o_{it}, o_{obj[]}\}$ — o_r creates object of `ProductCatalog` which creates o_{ps1}, o_r also creates o_s, then o_s creates o_{sli} and o_v, and o_v creates o_{it} and $o_{obj[]}$.

createClosure(o_i) is an upper bound on the nodes in *Boundary*(o_i); intuitively, an object o_j in *createClosure*(o_i) stays in the boundary until (roughly) o_j flows to an "outside" object o_k.

Object Flow Another important observation is the flow of objects. Let o_i^r be a run-time object that has access to another object, o_j^r — that is, there is an access edge $o_i^r \rightarrow o_j^r$ in the current run-time object graph. Object o_j^r can flow from o_i^r to

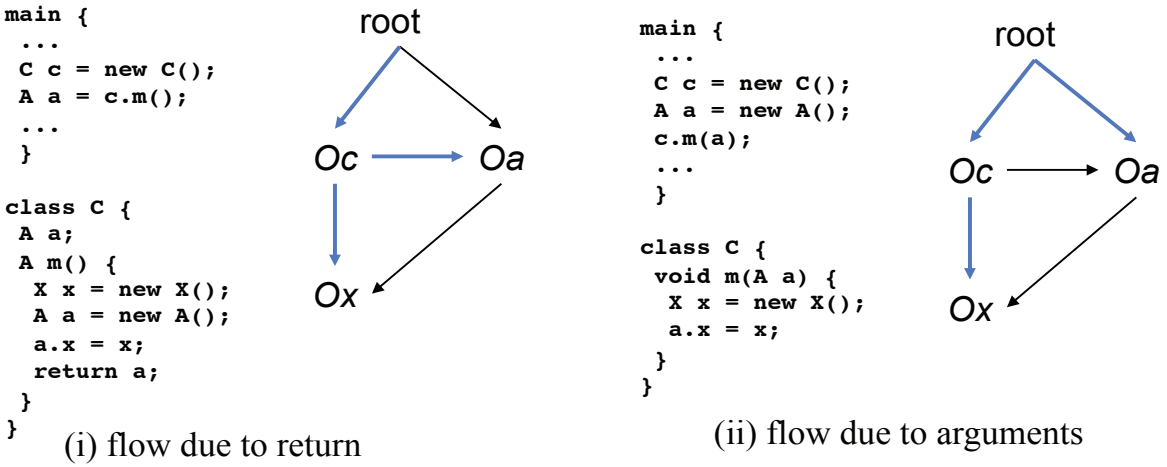

Figure 4.6: Object flows. Blue (thick) edges denote *create* edges.

another object, o_k^r, in one of two cases: (i) due to a return: o_j^r is returned from o_i^r to o_k^r, or (ii) due to an argument: o_j^r is passed as an argument from o_i^r to o_k^r. Below, we describe the two cases:

(i) Flow due to return. Recall that o_i^r has access to o_j^r — i.e., there is edge $o_i^r \rightarrow o_j^r$ in the current object graph. Object o_j^r flows from o_i^r to o_k^r due to a return if we have (1) o_k^r has access to o_i^r, i.e., there is edge $o_k^r \rightarrow o_i^r$, and (2) method m invoked on receiver o_k^r executes statement $l = r.n()$ where r points to o_i^r, and l points to o_j^r (i.e., o_j^r is returned from o_i^r to o_k^r due to statement $l = r.n()$)[9]. *Og* reflects this flow by *edge triple* $o_k \rightarrow o_i$, $o_i \rightarrow o_j$ and $o_k \rightarrow o_j$.

(ii) Flow due to arguments. Again, we have $o_i^r \rightarrow o_j^r$. Object o_j^r flows from o_i^r to o_k^r due to arguments if we have (1) $o_i^r \rightarrow o_k^r$, and (2) method m invoked on receiver o_i^r executes statement $r.n(r_1)$ where r points to o_k^r and r_1 points to o_j^r (i.e., o_j^r is passed from o_i^r to o_k^r as an argument in statement $r.n(r_1)$). *Og* reflects this flow by edge triple $o_i \rightarrow o_k$, $o_k \rightarrow o_j$ and $o_i \rightarrow o_j$.

Consider Figure 4.6(i), which illustrates case (i). We have that `root` has access to o_c (`root` creates o_c) and o_c has access to o_a (again, o_c creates o_a). Subsequently statement `a = c.m()` in `main` returns o_a to `root` which results in an access edge from

[9]For brevity, we mention statement kind $l = r.n()$ only; the other statements that result in *out* edges, namely $l = r.f$, and $l = r[i]$ can be executed as well.

```
class Demo {
  public static void main(String[] args)
  {
1   Demo d = new Demo();   //o_demo
2   d.testA(args.length > 0);
  }
  public void testA(boolean b) {
    A a;
3   a = new A(b);          //o_a
  }
}
class A {
  boolean mod;
  B b;
  A(boolean m) {
4   mod = m;
5   b = new B(this);       //o_b
  }
  void off() {
6   mod = false;
  }
}
```

```
class B {
  C c;
  D d;
  B(A a) {
7   c = new C(a);//o_c
8   d = new D(); //o_d
  }
}

class C {
  A a;
  C(A na) {
9   a = na;
10  if (a.mod) { a.off(); }
  }
}

class D {
  int i;
  D() {
10  i = 0;
  }
}
```

Figure 4.7: An Example of Invalid Triple.

`root` to o_a. The flow of o_a from o_c to `root` is reflected by edge triple `root`$\rightarrow o_c$, $o_c \rightarrow o_a$, `root`$\rightarrow o_a$. Now consider Figure 4.6(ii) which illustrates case (ii). We have that `root` accesses o_c and o_a (`root` creates both objects), and statement `c.m(a)` passes o_a to o_c which results in an access edge from o_c to o_a. The flow of o_a from `root` to o_c is reflected by edge triple `root` $\rightarrow o_c$, $o_c \rightarrow o_a$, `root`$\rightarrow o_a$ (note that this triple is identical to the triple in case (i), but it represents different run-time semantics).

In Figure 4.5 edge triple $o_r \rightarrow o_{ps1}$, $o_r \rightarrow o_s$, $o_s \rightarrow o_{ps1}$ represents the fact that a `ProductSpec` object flows into a `Sale` object as argument from the `Register` object.

The notion of the edge triple is central to the analysis; the tracking of flow of objects is at the heart of the precise computation of dominance boundary information. From now on, we will interchangeably denote an edge triple as a triple of edges, or as an ordered triple of nodes. An edge triple $o_k \rightarrow o_i$, $o_i \rightarrow o_j$, $o_k \rightarrow o_j$ is denoted as an ordered triple of nodes as follows: o_k, o_i, o_j.

Valid Triple Yet another important observation is that not every edge triple $o_k \rightarrow o_i, o_i \rightarrow o_j, o_k \rightarrow o_j$ represents valid object flow. Suppose that edges $o_i \rightarrow o_j$

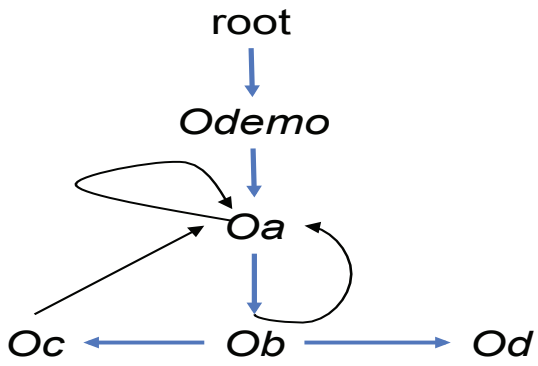

Figure 4.8: Object graph for Example in Figure 4.7. Blue edges denote *create* edges.

and $o_k \rightarrow o_j$ are due to object creation (lines 1-2 in the object graph construction in Figure 4.1) and $o_i \rightarrow o_k$ is due to inflow (lines 5-6). Clearly, edges $o_i \rightarrow o_j$ and $o_k \rightarrow o_j$ refer to two distinct run-time objects that are represented with the same name, o_j.

For example, consider triple o_c, o_a, o_a from the graph in Figure 4.8 for code in Figure 4.7 (this triple involves self edge $o_a \rightarrow o_a$). It is easy to see that this triple does not represent valid flow: there is no flow from o_a to o_c due to a return, and there is no flow from o_c to o_a due to arguments. The only triple that involves $o_a \rightarrow o_a$ and represents valid flow is o_a, o_b, o_a: o_a accesses o_b, o_a accesses itself through `this`, and o_a passes itself to o_b as an argument in `new B(this)`.

The analysis uses predicate $validTriple(o_k, o_i, o_j)$ to filter out invalid triples. Predicate *validTriple* is implemented by recording the statement that causes the creation of an edge; $validTriple(o_k, o_i, o_j)$ examines a triple o_k, o_i, o_j and checks the two cases: (i) if $o_k \rightarrow o_j$ is an *out* edge, and there is a statement $l = r.n()$ associated to this edge such that $o_i \in Pt(r)$, $validTriple(o_k, o_i, o_j)$ returns true; (ii) if $o_i \rightarrow o_j$ is an *in* edge and there is a statement $l.n(r)$ in method m associated to it such that $o_k \in Pt(\texttt{this}_m)$, $validTriple(o_k, o_i, o_j)$ returns true as well; otherwise, $validTriple(o_k, o_i, o_j)$ returns false. The predicate increases the memory needed to store the object graph; however, the impact of *validTriple* on scalability and precision is significant — in fact, without it, the analysis does not scale even to a relatively small program.

procedure `computeBoundary`
uses Og, *createClosure*, *validTriple*, *isOutside*
input o_i
output $Boundary(o_i)$
[1] $Out = \{o_j \mid isOutside(o_i \rightarrow o_j)\}$
[2] $In = createClosure(o_i)$ - Out
[3] $W = \{o_1 \rightarrow o_2 \mid o_1 \in In \wedge o_2 \in Out\}$

[4] while $W \neq \emptyset$
[5] remove $o \rightarrow o_j$ from W, mark it as visited
[6] if o_j is not visited $\wedge$ $o_j \in createClosure(o_i)$
[7] mark o_j as visited
[8] remove $createClosure(o_j)$ from In
[9] add $createClosure(o_j)$ to Out
[10] foreach $o_{k'} \in createClosure(o_j)$
[11] foreach *create* edge $o_{k''} \rightarrow o_{k'}$ s.t. $o_{k''} \in In$
[12] if $o_{k''} \rightarrow o_{k'}$ is not visited, add $o_{k''} \rightarrow o_{k'}$ to W
[13] foreach $validTriple(o, o_j, o_k)$
[14] remove o_k from In; add o_k to Out;
[15] if $o \rightarrow o_k$ is not visited, add $o \rightarrow o_k$ to W
[16] foreach $validTriple(o, o_{k'}, o_j)$ s.t. $o_{k'} \in In$
[17] if $o_{k'} \rightarrow o_j$ is not visited, add $o_{k'} \rightarrow o_j$ to W
[18] foreach $validTriple(o_{k'}, o, o_j)$ s.t. $o_{k'} \in In$
[19] if $o_{k'} \rightarrow o_j$ is not visited, add $o_{k'} \rightarrow o_j$ to W

[20] $Boundary(o_i) = \{o \rightarrow o_j \in Og \mid o \in In \wedge o_j \in In\}$.

procedure `isOwned`
input $o_i \rightarrow o_j \in Og$
output boolean: whether $o_i \rightarrow o_j$ is owned by o_i
if $o_i \rightarrow o_j \in Boundary(o_i)$ **return true**
else **return false**

Figure 4.9: Computes the boundary of o_i and checks if $o_i \rightarrow o_j$ is owned.

The analysis makes use of predicate $isOutside(o_i \rightarrow o_j)$. $isOutside(o_i \rightarrow o_j)$ returns true if there exists o_k such that o_k, o_i, o_j is a valid triple — that is, there exists some "outside" o_k such that either (i) o_j is returned from o_i to o_k, or (ii) o_j is passed as an argument from o_k to o_i; as a result, there could be a path to o_j through o_k (and not through o_i) in which case o_i may not dominate o_j.

Analysis Description The analysis is presented in Figure 4.9. The proce-

dure `computeBoundary` computes the dominance boundary of o_i. Then procedure `isOwned` checks an edge $o_i \rightarrow o_j \in Og$ to see whether it is owned by o_i, based on the dominance boundary computed by `computeBoundary`. If the procedure `isOwned` returns true for every field edge $o_i \xrightarrow{f} o_j$, the ownership analysis concludes that the association through f is `owned`.

To compute the dominance boundary, the analysis maintains sets *Out*, *In*, and worklist W. Set *Out* contains the current set of "outside" objects accessible to the boundary. These are objects that either (i) flow to the "outside" *from* the boundary of o_i, or (ii) they flow *to* the boundary of o_i from "outside". Set *Out* is initialized to the set of objects o_j such that $isOutside(o_i \rightarrow o_j)$ is true (line 1). The initial set *Out* captures the objects o_j such that one of the following is true: (i) o_j is directly returned from o_i (e.g., through a statement such as `a = c.m()` in Figure 4.6(i) which causes edge $o_c \rightarrow o_a$ to be an outside edge, and o_a to be in the initial *Out*), or (ii) o_j is passed from outside into o_i as an argument (e.g., through a statement such as `c.m(a)` in Figure 4.6(ii) which causes edge $o_c \rightarrow o_a$ to be an outside edge and o_a to be in the initial *Out*). Set *In* contains the current (over) approximation of the dominance boundary; it is initialized to $createClosure(o_i)$ minus the objects returned from o_i, i.e, $createClosure(o_i) - Out$ (line 2). Worklist W contains the set of cut edges — edges between the boundary *In* and the outside objects *Out* (line 3).

The analysis starts with initial sets *In*, *Out* and W and proceeds to identify all objects, originally in *In*, that are reachable from the initial *Out*. For every new edge $o \rightarrow o_j$ ($o \in In$ is an "inside" object, and $o_j \in Out$ is an "outside" object) taken from the worklist, the analysis does three things.

First, it examines o_j (lines 6-12). If o_j is in $createClosure(o_i)$ and was found to be in *Out*, the analysis removes the entire $createClosure(o_j)$ from *In* and adds it to *Out* (lines 8-9). Clearly, o_j is reachable from the "outside", then the objects reachable on *create* edges from o_j are also reachable from the "outside". Next, the analysis identifies *create* edges whose source $o_{k''}$ is in *In* and target $o_{k'}$ was just found to be in *Out*, and adds these edges to W(lines 10-12).

Second, the analysis identifies objects o_k, such that o, o_j, o_k is a valid triple —

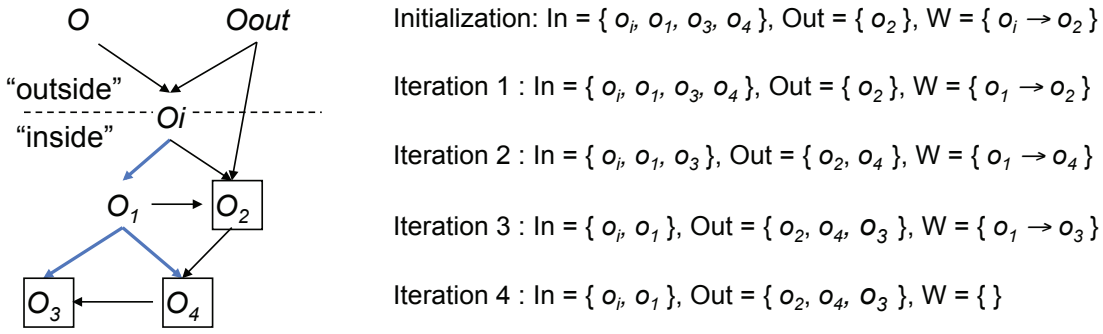

Figure 4.10: Boundary computation example. The boxed objects are found to be in *Out*.

in other words, we may have that o_k flows from o to "outside" o_j, or o_k flows from "outside" o_j to o. The analysis adds o_k to *Out* and $o \rightarrow o_k$ to W (lines 13-15). If o_k was in *In* until this point (i.e., o_k was an "inside" object, until it was passed to "outside" object o_j), when the edge is removed from W, o_k's *createClosure* will be removed from *In* and added to *Out*.

Third, the analysis identifies objects $o_{k'}$ in *In* such that "outside" object o_j flows to $o_{k'}$ (lines 16-19); this may cause an object deeper in the boundary to become reachable from outside, which will be discovered when edge $o_{k'} \rightarrow o_j$ is examined at line 5 in a subsequent iteration of the while loop. The analysis terminates because each object graph edge appears in W at most once.

Examples Consider the object graph in Figure 4.5, and consider the computation of the boundary of o_r. *In* is initialized to $createClosure(o_r) = \{o_r, o_{ps1}, o_s, o_{sli}, o_v, o_{it}, o_{obj[]}\}$. $Out = \{o_m\}$. $W = \{o_r \rightarrow o_m, o_s \rightarrow o_m\}$. The iterations at lines 6-19 for the two edges in W do not add new edges to W, and do not change *In* nor *Out*. Thus $Boundary(o_r)$ consists of nodes o_{ps1}, o_s, o_{sli}, o_v, o_{it}, $o_{obj[]}$ and the edges between them.

As another example, consider Figure 4.10. "Outside" object o_2 is passed as an argument to o_i. o_i then passes o_2 to "inside" object o_1; o_1 then passes "inside" object o_4 to o_2 and o_4 becomes "outside"; o_1 passes "inside" object o_3 to o_4 and o_3 becomes "outside". The workings of the analysis are shown in Figure 4.10.

Consider the example object graph in Figure 4.3 for code in Figure 4.4. As

described at the beginning of this Section, a standard dominator algorithm will determine that `Y`'s container, o_{cy}, does not dominate its array, o_d. In contrast, our analysis computes initially $In = createClosure(o_{cy}) = \{o_{cy}, o_d\}$, $Out = \{o_{z2}\}$ and thus W is empty. Then the analysis determines that $Boundary(o_{cy})$ equals $\{o_{cy} \rightarrow o_d\}$. Therefore, we have that o_{cy} dominates its array o_d.

4.3.3 Complexity

Let N be the size of the program being analyzed—that is, the number of statements, the number of object names and the number of variables is of order N. The complexity of the underlying Andersen-style points-to analysis is $O(N^3)$ [13].

The complexity of the ownership client is dominated by the ownership inference in Figure 4.9. In procedure `computeBoundary`, for each object o_i, create edge $o_{k''} \rightarrow o_{k'}$ is processed once at lines 10-12. Thus there are at most $O(N^2)$ create edges $o_{k''} \rightarrow o_{k'}$ processed at lines 10-12. For each edge $o \rightarrow o_j$ in W, there are at most $O(N)$ o_k objects processed at lines 13-15, and at most $O(N)$ $o_{k'}$ objects processed at lines 16-19. There are at most $O(N^2)$ $o \rightarrow o_j$ edges processed in W, thus the running time for procedure `computeBoundary` is $O(N^4)$. In procedure `isOwned`, there is $O(N)$ work for each edge $o_i \rightarrow o_j$, and there are at most $O(N^2)$ edges, thus the running time is $O(N^3)$. Therefore the complexity of ownership inference is $O(N^4)$.

4.4 Immutability Analysis

4.4.1 Immutability Inference

The immutability inference is presented in Figure 4.11. Procedure `computeModVars` from Lines 0-5 performs standard side-effect analysis [14, 15] which computes a *Mod* set for each method m. Lines 0-1 process each statement s: $p.f = q$ and store p in the *Mod* set for the enclosing method of s. Subsequently, lines 2-5 propagate the *Mod* sets backwards on the call graph. Set $Mod(m)$ contains all reference variables p on the left-hand side of an instance field write, reachable on a call chain from m. The union of the points-to sets of these variables approximates the set of objects that may be modified during the invocation of m.

procedure **computeModVars**
input Pt: $R \rightarrow \mathcal{P}(O)$
output Mod: $m \rightarrow \mathcal{P}(R)$
[0] foreach instance field write s: $p.f = q$
where `p`$\neq$`this` OR $EnclMethod(s)$ is not a constructor
[1] add p to $Mod(EnclMethod(s))$
[2] while changes occur in Mod
[3] foreach call s: $C.m()$ or $r.m()$
[4] foreach target m' of the call
[5] add $Mod(m')$ to $Mod(EnclMethod(s))$

procedure **isReadOnly**
input $o_i \rightarrow o_j \in O \times O$ $Mod : m \rightarrow \mathcal{P}(R)$
output readOnly: boolean
[6] foreach call s: $r.m(...)$ s.t. $r \neq$ `this` and $o_i \in Pt(r)$
[7] $modSet(o_i) = modSet(o_i) \cup Pt(Mod(target(o_i, m)))$
[8] if $TrClosure(o_j) \cap modSet(o_i) \neq \emptyset$
[9] return false
[10] return true;

Figure 4.11: Immutability inference: computing the read-only status.

Finally, lines 6-9 take an edge $o_i \rightarrow o_j \in Og$ as input and attempt to show that for all run-time edges $o_i^r \rightarrow o_j^r$ represented by this edge, o_i^r has read-only access to o_j^r. The analysis examines each method call $r.m(...)$ on receiver o_i (i.e., $o_i \in Pt(r)$). $TrClosure(o_j)$ denotes the transitive closure of o_j on the points-to graph—that is, the set of all nodes reachable from o_j on a path of field edges. $Pt(S)$ extends the Pt notation over sets as follows: $Pt(S) = \bigcup_{p \in S} Pt(p)$. If for some call the transitive closure of o_j intersects with the set of modified objects of the run-time target of the call (i.e., $target(o_i, m)$), the analysis determines that edge $o_i \rightarrow o_j$ is not immutable. If this intersection is always empty, the analysis determines that $o_i \rightarrow o_j$ is immutable.

In the Point-of-Sale code method `getTotal` in `Sale` iterates over the collection of `SaleLineItem`s and calls `getSubtotal` on each `SaleLineItem` object. The body of method `getSubtotal` is as follows:

```
return spec.getPrice().times(quantity);
```

We have that field *spec* of o_{sli} points to o_{ps1} and field *price* of o_{ps1} points to o_{m1} (o_{m1}

```
class A {
   B b1;
   A(B _b1) { b1 = _b1; ...  }
   m() { B b2 = new B(); b2.setField(10); }
}
class B() {
   int field;
   void setField(int number) { field = number; }
}
main() {
   B b1 = new B(); b1.setField(5);
   A a = new A(b1); a.m();
}
```

Figure 4.12: Imprecision of immutability inference.

represents the `Money` object that holds the price of the product). Thus, `getSubtotal` calls method `times` on o_{m1}. The analysis correctly determines that $Mod(\texttt{times})$ equals $\{\texttt{this}_{times}\}$—that is, `times` *changes* the value of the receiver object. Thus, $Mod(\texttt{getSubtotal})$ equals $\{\texttt{this}_{times}\}$ and we have that o_{m1} is included in set $Pt(Mod(\texttt{getSubtotal}))$.

Consider the call to method `getSubtotal` in `getTotal` in `Sale`, and the effect of this call on edge $o_{sli} \overset{spec}{\rightarrow} o_{ps1}$. The intersection of the set of objects modified by `getSubtotal` and the transitive closure of o_{ps1} is non-empty; it includes o_{m1}. The analysis determines that a `SaleLineItem` object can modify a `ProductSpec` object which is a violation of the immutability constraint in Figure 2.1. Further examination revealed that this was a serious bug in the code in [5], shown at line 25 in Appendix A; it caused subsequent sales to fetch wrong product prices and compute incorrect totals.

If the procedure for checking an edge returns true for every edge labeled with f, the immutability analysis concludes that the association through f is `read-only`.

4.4.2 Improved Immutability Inference

The algorithm in Figure 4.11 may incur substantial imprecision. Consider the code in Figure 4.12. Field $b1$ is immutable in A. The B object created in

`main` and referred to by field $b1$ is denoted by name o_{b1}, and the B object created in `m` is denoted by o_{b2}. Mod(`setField`) equals $\{\texttt{this}_{setField}\}$; it is propagated to Mod(`m`) and we have that Mod(`m`) equals $\{\texttt{this}_{setField}\}$ as well. The points-to set of $\texttt{this}_{setField}$ contains both o_{b1} and o_{b2} and the analysis concludes imprecisely that $b1$ is mutable.

To improve the analysis we introduce a limited form of context sensitivity. When propagating the Mod set of the callee (line 5 in Figure 4.11), the analysis "maps" modified formal parameters to their corresponding actuals. More precisely, it examines every variable $v \in Mod(m')$. If v is an unassigned formal parameter of m', v is mapped to the corresponding actual at the call and the actual is added to $Mod(EnclMethod(s))$; otherwise v itself is added to $Mod(EnclMethod(s))$. Consider again the code in Figure 4.12. When propagating Mod(`setField`) to Mod(`m`) the analysis maps $\texttt{this}_{setField}$ to the actual argument at the call, namely variable `b2`. As a result Mod(`m`) equals {`b2`}. Since `b2` points to o_{b2} only, the intersection of the transitive closure of o_{b1} and $\{o_{b2}\}$ is empty and the analysis concludes that $b1$ is immutable in A.

4.4.3 Complexity

Again, Let N be the size of the program being analyzed—that is, the number of statements, the number of object names and the number of variables is of order N. The complexity of the underlying Andersen-style points-to analysis is $O(N^3)$ [13].

The complexity of the immutability client is dominated by the checking of edges (lines 6-9 in Figure 4.11). The computation of the transitive closures of all nodes is $O(N^3)$. For each o_i, the analysis processes at most $O(N)$ calls. For each call it takes $O(N)$ time to compute $modSet(o_i)$ at lines 6-7. Thus it takes total of $O(N^3)$ time processing at lines 6-7. For each edge $o_i \rightarrow o_j$ the analysis does $O(N)$ work at line 8 checking whether each $o' \in modSet(o_i)$ is in $TrClosure(o_j)$. Since there are $O(N^2)$ edges, the complexity of immutability inference is $O(N^3)$.

- **Conditional** i: $if(l)$ {...}, $while(l)$ {...}.
  Implicit flow edge: $l \rightsquigarrow s_i$.
  Implicit flow edge: $s_{immed\_encl} \rightsquigarrow s_i$
  or $m_{encl} \rightsquigarrow s_i$.
- **Assignment** $l = (...operator)\ r$.
  Explicit flow edge: $r \rightsquigarrow l$.
  Implicit flow edge: $s_{immed\_encl} \rightsquigarrow l$
  or $m_{encl} \rightsquigarrow l$.
- **Instance field write** $l.f = r$.
  Explicit flow edge: $r \overset{*}{\rightsquigarrow} l.f$.
  Implicit flow edge: $s_{immed\_encl} \overset{*}{\rightsquigarrow} l.f$
  or $m_{encl} \overset{*}{\rightsquigarrow} l.f$.
  $\forall l'$ (l' is alias of l), explicit flow edge: $l.f \overset{*}{\rightsquigarrow} l'.f$.
- **Instance field read** $l = r.f$.
  Explicit flow edge: $r.f \rightsquigarrow l$.
  Implicit flow edge: $s_{immed\_encl} \rightsquigarrow l$
  or $m_{encl} \rightsquigarrow l$.
- **Method call** i: $l = r_0.m(r_1, ...)$.
  $\forall m'(\texttt{this}, p_1, ..., ret)$ (runtime target),
  Explicit flow edges:
  $r_0 \overset{(_i}{\rightsquigarrow} \texttt{this}_{m'}$, $r_1 \overset{(_i}{\rightsquigarrow} p_1$, ... $ret \overset{)_i}{\rightsquigarrow} l$.
  Implicit flow edges:
  $s_{immed\_encl} \overset{(_i}{\rightsquigarrow} \texttt{this}_{m'}$, $s_{immed\_encl} \overset{(_i}{\rightsquigarrow} p_1$,...
  $s_{immed\_encl} \overset{(_i}{\rightsquigarrow} m'$, $s_{immed\_encl} \rightsquigarrow l$
  or
  $m_{encl} \overset{(_i}{\rightsquigarrow} \texttt{this}_{m'}$, $m_{encl} \overset{(_i}{\rightsquigarrow} p_1$, ...
  $m_{encl} \overset{(_i}{\rightsquigarrow} m'$, $m_{encl} \rightsquigarrow l$.

Figure 4.13: Construction of flow graph.

4.5 Information Flow Analysis

The information flow analysis consists of three parts: generation of annotated flow graph, summarization of the effects of callees on callers, and demand-driven reachability propagation on the summarized graph. This analysis is based on CFL-reachability [16], and builds on ideas from [17].

4.5.1 Construction of Flow Graph $\mathcal{FG}_0$

The flow graph $\mathcal{FG}_0$ has several kinds of nodes for capturing explicit and implicit flows: variable nodes (e.g., r), field dereference nodes (e.g., $r.f$), method nodes (e.g., m) that break down inter-procedural implicit flow into intra-procedural flows, and conditional statement nodes (e.g., s) that break down implicit flows which cross nested conditionals.

The edges in $\mathcal{FG}_0$ represent direct flows: flows that could not be broken down into smaller flows. There are several kinds of direct flow edges: (1) $l \rightsquigarrow r$ which represents flow from variable l into variable r, (2) $l \rightsquigarrow r.f$, which represents flow from variable l into field f of an object referred to by r, (3) $l.f \rightsquigarrow r$ which represents flow from field f of an object referred to by l into variable r, (4) $l \rightsquigarrow s$, which represents the relation of conditional variable l and the associated conditional statement node s, (5) $s \rightsquigarrow r$ or $s \rightsquigarrow r.f$, which represents the immediate control dependence of r (or $r.f$) on the associated conditional statement of s, (6) $m \rightsquigarrow r$ or $m \rightsquigarrow r.f$, when r (or $r.f$) is assigned without any immediate control dependence within the enclosing method m, (7) $s_i \rightsquigarrow s_j$ which connects the implicit flows across nesting conditionals, and (8) $s \rightsquigarrow m$, or $m \rightsquigarrow s$, which represents the effect of inter-procedural flows. We use notation $\rightsquigarrow$ to distinguish from notation $\mapsto$; $\rightsquigarrow$ denotes analysis flow (i.e., the representation of run-time flow), while $\mapsto$ denotes run-time flow. Below we describe the processing of each program statement kind.

The construction of flow graph $\mathcal{FG}_0$ is illustrated in Figure 4.13. When building $\mathcal{FG}_0$ the context-sensitive analysis annotates edges with certain information. The summarization and subsequent reachability propagation take these annotations into account and filter out certain infeasible flow paths. Below we describe the flow edges constructed in this stage.

4.5.1.1 Explicit Flow Edges

The explicit flow edges are straight-forward. For each direct assignment, instance field write, and instance field read statement,[10] the data is transmitted from the right-hand-side expression variables to the left-hand-side variable. For example,

[10]Each static field could be considered as a variable, thus static field reads and writes could be represented by direct assignments

for direct assignments $l = r$, there is explicit flow edge $r \rightsquigarrow l$. Flows through fields need to be captured in a special way in order to account for aliasing. For example, flows $l_1 \rightsquigarrow v_1.f$ and $v_2.f \rightsquigarrow l_2$ where v_1 and v_2 are aliases (i.e., they could point to the same object at run-time), lead to flow $l_1 \rightsquigarrow l_2$. Therefore, whenever a field write $l.f = r$ occurs, explicit flow edges are generated between the right-hand-side r and all written object fields $l'.f$, where l' is an alias of l. Instead of building an edge $r \overset{*}{\rightsquigarrow} l'.f$, we break it into edges $r \overset{*}{\rightsquigarrow} l.f$, and $l.f \rightsquigarrow l'.f$. This is done in order to enable a precision improvement which is explained in Section 4.5.2. For method call statement, each run-time method invocation has the effect of generating explicit flow from actual arguments to the corresponding instances of the formal parameters, as well as generating explicit flow from the instance of the return variable to the variable on the left-hand-side of the method call.

The handling of assignments and method calls directly follows the information flow model defined in Section 3.4. The handling of field accesses uses alias information instead of points-to information; we believe that this representation is clearer than representation with a set of objects, and as mentioned in previous paragraph, it is more informative when tracking indirect information flow to improve precision explained in Section 4.5.2.

4.5.1.2 Implicit Flow Edges

The implicit flow edges capture information transmitted due to control flow. We break down inter-procedural implicit flows and complex intra-procedural flows into several smaller connecting flows.

For each conditional statement, a node s_i is created to capture the flows through this conditional statement. For example, in Figure 4.14, suppose node s_8 represents the *if* statement at line 8 and node s_7 represents the *if* statement at line 7. For each variable tested at s_8, we generate implicit flow edges. There are implicit flow edges `tries` $\rightsquigarrow s_8$, `i` $\rightsquigarrow s_8$ and `secret` $\rightsquigarrow s_8$. These edges illustrate that flow transmitted through statement s_8 transfers the information from variables `tries`, `i` and `secret`.

For each conditional statement s_i, we check if there exists a conditional state-

```
[1]   public class GuessANumber {
[2]    int secret;
[3]    int tries;
       ...
[4]    void makeGuess ( Integer num )
         throws NullPointerException
       {
[5]       int i = 0;
[6]       if ( num != null ) i = num.intValue();
[7]       if ( i >= 1 && i <= 10 ) {
[8]           if ( tries > 0 && i == secret ) {
[9]             tries = 0;
[10]            finishApp("You win!");
              }
[11]          else {
[12]            tries-- ;
[13]            if ( tries > 0 )
[14]              message.setText("Try again");
[15]            else
[16]              finishApp("Game over!");
              }
          }
[17]      else message.setText("Out of range");
        }
    }
```

Figure 4.14: Guess-a-Number web application

ment s immediately enclosing s_i. If there exists such a statement, we build an implicit flow edge from s to s_i. For example, in Figure 4.14, s_8 at line 8 has an immediately enclosing conditional statement, namely s_7, the *if* statement at line 7. The implicit flow edge $s_7 \rightsquigarrow s_8$ shows that flow through s_8 carries information from s_7 as well. Thus, the implicit flow across nested conditionals s_7 and s_8, such as the flow `i` $\rightsquigarrow$ `tries`, is captured by connecting several flows: `i` $\rightsquigarrow s_7$ (when processing statement at line 7), $s_7 \rightsquigarrow s_8$ (what we just described), and later another implicit flow $s_8 \rightsquigarrow$ `tries` (as will be described below).

For each assignment, instance field read, and instance field write, we check if there exists a conditional statement s immediately enclosing the assignment. If there exists such a conditional, we build an implicit flow edge from s to the left-

hand-side variable of the assignment. In Figure 4.14, the assignment at line 9 has an immediately enclosing *if* statement, s_8 at line 8. An implicit flow edge $s_8 \rightsquigarrow$ `tries` shows that the assignment of variable `tries` is affected by the control transfer at s_8. Similarly, the implicit flow `secret` $\rightsquigarrow$ `tries` is captured by connecting flow edges `secret` $\rightsquigarrow s_8$ (when processing line 8) and $s_8 \rightsquigarrow$ `tries` (described here).

Method call statements are processed similarly to assignment statements. Each formal parameter p_i is considered as assigned. Thus, implicit flow edges are constructed connecting the immediately enclosing conditional s and p_i. Another implicit flow edge, $s \rightsquigarrow m'$, is constructed, to capture the inter-procedural flow from the current method to the target method m'. In Figure 4.14, line 14 calls method `message.setText`; this call is affected by the *if* statement at line 13, and thus we have $s_{13} \rightsquigarrow m_{setText}$ which illustrates that flow transmitted to method $m_{setText}$ transfers the information from conditional s_{13}.

In the above statements, when they have no immediately enclosing conditional s in the enclosing method m, implicit flow edges from method node m are constructed. Consider the code for method `message.setText`:

```
   static void setText ( String msg ) {
[18]  StringBuffer text;
[19]  text = new StringBuffer("GuessNum:  ");
[20]  text.append(msg);
      ...
   }
```

The assignment at line 19 has no immediately enclosing conditional statement in `setText`. However, method `message.setText` is called at line 14 in Figure 4.14, and therefore it is affected by the conditional statement s_{13}. Thus, the flow effect of s_{13} on variable `text` needs to be captured. At line 19, implicit flow edge $m_{setText} \rightsquigarrow$ `text` is constructed, which connects with the edge $s_{13} \rightsquigarrow m_{setText}$ to capture the inter-procedural flow $s_{13} \rightsquigarrow$ `text`.

4.5.1.3 Conditional Scope

When constructing implicit flow edges, it is necessary to obtain the immediately enclosing conditional statement. In other words, it is necessary to define the scope of conditional statements.

As inter-procedural implicit flow is broken down into intra-procedural flow, the scope of a conditional statement could be identified intra-procedurally. The scope identification is based on the intra-procedural control flow graph (CFG).[11] Each node in the graph represents a basic block, i.e., several statements without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow.

Block b is the *exit* of a conditional statement s of block a, if and only if b *post-dominates* a (such that every path from a to the method exits contains b). The successor blocks of b are not controlled by conditional statement s. Thus, the conditional statement s_j immediately enclosing s can be found by computing the post-dominance relations in the CFG.

4.5.1.4 Flow Edge Annotations

A context-insensitive information flow analysis is bound to produce substantial imprecision, as well as overhead in analysis cost due to the tracking of infeasible flow. Therefore, there is a need for a context-sensitive analysis—that is, analysis that tracks flow through different contexts of invocation of a method precisely.

Imprecision of Context-insensitive Analysis. Consider the example in Figures 3.3 and 4.15 and let us be interested in the flow of constant `LOCFLG` defined in interface `ZipConstants`. The following edges created by the context-insensitive

[11] We work with a Jimple CFG in Soot, which is essentially the standard CFG.

```
void main() {
    ZipEntry ph_ZE;
    ZipInputStream ph_ZIS;
    long ph_long;
20  ph_ZE = new ZipEntry();
21  ph_ZIS = new ZipInputStream();
22  ph_ZE.setSize(ph_long);
23  ph_long = ph_ZE.getSize();
24  ph_ZE = ph_ZIS.getNextEntry();
}
```

Figure 4.15: Placeholder `main` method for `zip`.

analysis represent flow relevant to `LOCFLG`:

`LOCFLG` $\leadsto$ `get32.off`	(due to line 6)
`get32.off` $\leadsto$ `get16.off`	(line 14)
`get16.off` $\leadsto$ `get16.i1` $\leadsto$ `get16.ret`	(lines 12 and 13)
`get16.ret` $\leadsto$ `get32.i1` $\leadsto$ `get32.ret`	(lines 14,16-17)
`get32.ret` $\leadsto$ `readLOC.i2`	(line 8)
`readLOC.i2` $\leadsto$ $\texttt{this}_{\texttt{getSize}}\texttt{.size}$	(line 9 and aliasing)
$\texttt{this}_{\texttt{getSize}}\texttt{.size}$ $\leadsto$ `getSize.ret`	(line 19)
`getSize.ret` $\leadsto$ `ph_long`	(line 23 in `main`)

For clarity the flows due to code lines 14, 16 and 17 are simplified. These statements result in flow edges `get16.ret` $\leadsto$ `get32.i1` $\leadsto$ `get32.i3` $\leadsto$ `get32.ret` but for clarity we omit variable `get32.i3`. Given these direct flows it is easy to see that the analysis infers flow from `LOCFLG` to `ph_long` and thus it reports that `LOCFLG` could flow to client code. In fact, this is infeasible flow that is due to the context-insensitive handling of method `get32`. Flow edge `LOCFLG` $\leadsto$ `get32.off` results from the call of method `get32` at line 6, and flow edge `get32.ret` $\leadsto$ `readLOC.i2` results from the return from `get32` at line 8. Clearly, this sequence represents an invalid flow path.

Context-sensitivity. When building flow graph $\mathcal{FG}_0$ as described in Figure 4.13, our context-sensitive analysis annotates edges with certain information.

The summarization and subsequent reachability propagation take these annotations into account and filter out infeasible paths.

There are no annotations on the flow edges generated for assignments and instance field reads. The parentheses annotations at method calls are standard CFL-reachability annotations: they denote flow into context copies of formal parameters, and flow from context copies of return variables. Consider parenthesis $(_i$ in $r_1 \overset{(_i}{\rightsquigarrow} p_1$; it denotes flow from actual parameter r_1 to the instance of the formal parameter p_1 for call site i. Analogously, parenthesis $)_i$ in $ret \overset{)_i}{\rightsquigarrow} l$ denotes flow from the instance of return variable *ret* for call site i to the left-hand side of the call l. The parentheses are matched to form valid flow paths — for example, $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow ret \overset{)_i}{\rightsquigarrow} l_1$ is a valid path, but $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow ret \overset{)_j}{\rightsquigarrow} l_2$ is not.

The "wildcard" $*$ annotations at field writes handle flow through objects which transcends calling contexts. They allow information that flows into objects through a sequence of method calls to be returned to a different method. They are best explained by the following example. Suppose that there is a call site `i: r.set(k)` which sets field f of r to the value of k (i.e., there is statement `this.f=p;` in method `set`). Later there is a call `j: l=r.get()` which returns field `f` of `r` (i.e., there is statement `return this.f;`). The flow edges for these statements are: $\texttt{k} \overset{(_i}{\rightsquigarrow} \texttt{p} \overset{*}{\rightsquigarrow} \texttt{this}_{\texttt{get}}\texttt{.f} \rightsquigarrow \texttt{get.ret} \overset{)_j}{\rightsquigarrow} \texttt{l}$. In the above example $(_i$ is concatenated with the wildcard and it is canceled by the wildcard resulting in transitive flow edge $\texttt{k} \overset{*}{\rightsquigarrow} \texttt{get.ret}$. Subsequently, the wildcard cancels $)_j$ resulting in flow edge $\texttt{k} \overset{*}{\rightsquigarrow} \texttt{l}$, although the method call and method return belong to different call sites.

4.5.2 Summarization

Procedure *Summarize* in Figure 4.16 computes the summary flow graph $\mathcal{FG}^*$. Intuitively, this procedure computes the flow effects due to method calls. *Summarize* operates on a worklist of edges *WL*; the worklist is initialized to the set of edges in $\mathcal{FG}_0$ that have $(_i$ (i.e., open parenthesis) annotations. Lines 2-8 remove an edge e_1 from the worklist and process this edge accordingly. There are two cases. Lines 3-5 process the case when the annotation on edge e_1 is of kind $(_i$. In this case, the algorithm examines each edge e_2 which is a successor of e_1 and concatenates e_1 and

procedure **Summarize**
input $\mathcal{FG}_0$: flow graph
output $\mathcal{FG}^*$: summarized $\mathcal{FG}_0$
initialize $\mathcal{FG}^* = \mathcal{FG}_0$
$WL = \{v_1 \overset{a}{\rightsquigarrow} v_2 \in \mathcal{FG}_0 \text{ s.t. } a \text{ is } (_i \text{ annotation}\}$
[1] while $WL \neq \emptyset$ do
[2] remove e_1: $v_1 \overset{a_1}{\rightsquigarrow} v_2$ from WL
[3] if a_1 is an $(_i$ annotation
[4] foreach e_2: $v_2 \overset{a_2}{\rightsquigarrow} v_3 \in \mathcal{FG}^*$ do
[5] if $e_3 = concat(e_1, e_2) \notin \mathcal{FG}^*$ add e_3 to $\mathcal{FG}^*$ and WL
[6] else if a_1 is an empty or $*$ annotation
[7] foreach e_2': $v_0 \overset{a_2'}{\rightsquigarrow} v_1 \in \mathcal{FG}^*$ do
[8] if $e_3' = concat(e_2', e_1) \notin \mathcal{FG}^*$ add e_3' to $\mathcal{FG}^*$ and WL

Figure 4.16: Computation of summarized flow graph $\mathcal{FG}*$.

e_2. If this concatenation results in a new edge e_3, e_3 is added to $\mathcal{FG}^*$ and WL. Lines 6-8 process the case when the annotation on edge e_1 is empty or $*$. The algorithm examines each predecessor edge e_2', already added to $\mathcal{FG}^*$ and WL, and possibly taken off WL before e_1. It concatenates e_2' with e_1 and if this concatenation results in a new edge e_3', e_3' is added to $\mathcal{FG}^*$ and WL. It is important to note that operation *concat* produces an edge *only if* the first edge has a $(_i$ annotation, and the second edge has one of the following annotations: empty, $*$, or matching $)_i$; otherwise, there is no edge:

$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \rightsquigarrow v_3) = v_1 \overset{(_i}{\rightsquigarrow} v_3$$
$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{)_i}{\rightsquigarrow} v_3) = v_1 \rightsquigarrow v_3$$
$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{*}{\rightsquigarrow} v_3) = v_1 \overset{*}{\rightsquigarrow} v_3$$
$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{*}{\rightsquigarrow} \texttt{this}.f) = v_1 \overset{*}{\rightsquigarrow} v'.f$$
$$(v' \textit{ is alias of the receiver at call site } i)$$

Intuitively, procedure *Summarize* propagates $(_i$ annotations forward until they are matched with a corresponding $)_i$ or a $*$ annotation. If $(_i$ is matched with a corresponding $)_i$ annotation, the resulting edge with empty annotation reflects the information flow effect of the callee method called at call site i on the caller method which contains call site i. If $(_i$ is matched with a $*$ annotation, it is "canceled" by

the $*$ and the resulting edge carries the $*$ annotation. The $*$ annotation, needed to track non-trivial flow through object fields, essentially cancels calling context information.

The transcending context effect of wildcard annotation may result in imprecision. Consider the following example in Figure 4.17, where we have flow edges $\texttt{number} \overset{(2}{\rightsquigarrow} \texttt{set.num} \overset{*}{\rightsquigarrow} \texttt{this}_{\texttt{set}}\texttt{.f}$, $\texttt{this}_{\texttt{get}}\texttt{.f} \rightsquigarrow \texttt{get.ret} \overset{)5}{\rightsquigarrow} \texttt{dest}$, and $\texttt{this}_{\texttt{get}}$ is an alias of $\texttt{this}_{\texttt{set}}$ due to methods `set` and `get` invoked on $a2$ at call sites 4 and 5. The wildcard annotation on edge $\texttt{set.num} \overset{*}{\rightsquigarrow} \texttt{this}_{\texttt{set}}\texttt{.f}$ would cancel the method call context $(_2$ and return context $)_5$, resulting in invalid flow edge $\texttt{number} \rightsquigarrow \texttt{this}_{\texttt{set}}\texttt{.f} \rightsquigarrow \texttt{this}_{\texttt{get}}\texttt{.f} \rightsquigarrow \texttt{dest}$.

```
  class A {
      int f;
      set(int num) { f = num; }
      int get() { return f; }
  }
  main() {
      int number;
1      A a1 = new A();
2      a1.set(number);
3      A a2 = new A();
4      a2.set(10);
5      int dest = a2.get();
  }
```

Figure 4.17: Imprecision due to context canceling.

To improve precision, we separate the handling of $*$-annotated edges to $\texttt{this}.f$. When a call context $(_i$ is canceled by such a $*$ annotation, $\texttt{this}.f$ is replaced by $v'.f$ such that v' is an alias of v, the receiver of the method call. Here the call context helps to precisely capture the points-to information of the field references, which introduces object sensitivity that improves analysis precision. In the previous example in Figure 4.17, when call context $(_2$ is canceled by wildcard on edge $\texttt{set.num} \overset{*}{\rightsquigarrow} \texttt{this}_{\texttt{set}}\texttt{.f}$, $\texttt{this}_{\texttt{set}}\texttt{.f}$ is replaced by `a1.f` resulting in flow edge $\texttt{number} \rightsquigarrow \texttt{a1.f}$, because `a1` is the receiver of method call at call site line 2. Since $\texttt{this}_{\texttt{get}}\texttt{.f}$ is not an alias of $a1.f$, the analysis would filter the invalid flow

$\texttt{number} \rightsquigarrow \texttt{dest}$.

In our example of Figure 3.3, procedure *Summarize* produces new edges as follows. Initially edges $\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.off}$ and $\texttt{get32.off} \overset{(_{14}}{\rightsquigarrow} \texttt{get16.off}$ are added to worklist *WL*. Subsequently edge $\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.off}$ is taken off the worklist and processed without the addition of new edges. Edge $\texttt{get32.off} \overset{(_{14}}{\rightsquigarrow} \texttt{get16.off}$ is taken off the worklist and processed on lines 3-5. The concatenation on line 5 results in new edge

$$\texttt{get32.off} \overset{(_{14}}{\rightsquigarrow} \texttt{get16.i1}$$

which is added to $\mathcal{FG}^*$ and *WL*. This edge is then processed on lines 3-5 resulting in new edge

$$\texttt{get32.off} \overset{(_{14}}{\rightsquigarrow} \texttt{get16.ret}$$

which is added to $\mathcal{FG}^*$ and *WL*. This edge is processed on lines 3-5 and the concatenation of line 5 results in new edge

$$\texttt{get32.off} \rightsquigarrow \texttt{get32.i1}$$

which is added to $\mathcal{FG}^*$ and the worklist. Note that this edge results from concatenation with the matching $)_{14}$ annotation. It is processed on lines 6-8. The algorithm examines its predecessor edges and finds edge $\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.off}$ which was processed on the worklist earlier. The concatenation on line 8 results in new edge

$$\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.i1}.$$

Processing this edge results in new edge

$$\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.ret}.$$

Processing this edge does not result in new edges. Edges $\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.ret}$ and $\texttt{get32.ret} \overset{)_8}{\rightsquigarrow} \texttt{readLOC.i2}$ are not concatenated because indices 6 and 8 do not match—clearly, these edges correspond to flows due to different contexts of

procedure **Propagate**
input $\mathcal{FG}^*$: summarized graph s: source node
output $\mathcal{FG}_p$: flow path graph wrt s
initialize Add path-annotated edges from s to $\mathcal{FG}_p$ and WL
[1] while $WL \neq \emptyset$ do
[2] remove e_1: $s \overset{p}{\rightsquigarrow} v_1$ from WL
[3] foreach e_2: $v_1 \overset{a}{\rightsquigarrow} v_2 \in \mathcal{FG}^*$ do
[4] if $e_3 = concat'(e_1, e_2) \notin \mathcal{FG}_p$ add e_3 to $\mathcal{FG}_p$ and WL

Figure 4.18: Computation of shallow flow from s.

invocation of `get32`.

4.5.3 Propagation

Recall that we are interested in uncovering all nodes in the flow graph that could be reached from a node s on a valid flow path. The flow graph with the additional summary edges added due to *Summarize* does not explicitly show these paths. For example, in example code of Figure 3.3, there is valid flow from `LOCFLG` to `get16.i1`: `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.off` $\overset{(_{14}}{\rightsquigarrow}$ `get16.off` $\rightsquigarrow$ `get16.i1`.

Procedure *Propagate* in Figure 4.18 computes graph $\mathcal{FG}_p$. $\mathcal{FG}_p$ contains path edges from s that represent shallow flow from s. The path edges are annotated with special *path annotations* that reflect the structure of the valid flow path from s. There are two kinds of path annotations: *Call* and *nCall*.

The *Call* annotation denotes flow paths that end on a call sequence. In our example of Figure 3.3, there is a *Call* path `LOCFLG` $\overset{Call}{\rightsquigarrow}$ `get16.i1` on a call sequence $(_6(_{14}$. This flow is due to the call to `get32` from caller `readLOC` at line 6, and subsequently to the call to `get16` from caller `get32` at line 14.

The *nCall* annotation denotes paths that do not end on a call sequence. These paths could be (1) empty paths consisting of intraprocedural, or matching interprocedural flow, (2) paths that end on a return sequence (e.g., $)_{14})_8$), or (3) paths that end on a $*$. Consider the code in Figures 3.3 and 4.15. There is an *nCall* path `get32.i1` $\overset{nCall}{\rightsquigarrow}$ `getSize.ret` which is due to flow `get32.i1` $\overset{)_8}{\rightsquigarrow}$ `readLOC.i2`, followed by flow `readLOC.i2` $\overset{*}{\rightsquigarrow}$ `getSize.this.size`, followed by flow `getSize.this.size` $\rightsquigarrow$ `getSize.ret`.

Procedure *Propagate* in Figure 4.18 finds nodes v reachable from s; it adds a path edge from s to v to $\mathcal{FG}_p$ with the corresponding flow path annotation. For initialization it considers all edges in $\mathcal{FG}^*$ from s and adds the appropriate path edges to $\mathcal{FG}_p$. Edges of kind $s \overset{(_i}{\rightsquigarrow} v$ result in path edges $s \overset{Call}{\rightsquigarrow} v$. Edges of kinds $s \rightsquigarrow v$, $s \overset{*}{\rightsquigarrow} v$ and $s \overset{)_j}{\rightsquigarrow} v$ result in path edges $s \overset{nCall}{\rightsquigarrow} v$. Each path edge $s \overset{p}{\rightsquigarrow} v_1 \in \mathcal{FG}_p$ is concatenated with edges $v_1 \overset{a}{\rightsquigarrow} v_2 \in \mathcal{FG}^*$. If the concatenation results in a new path edge from s, namely e_3, e_3 is added to $\mathcal{FG}_p$ and WL.

It remains to define the concatenation operation $concat'$. We need to consider concatenation of each possible path annotation (i.e., (1) *Call* and (2) *nCall*), with each possible edge annotation (i.e., (1) empty, (2) $(_i$, (3) $)_j$ and (4) $*$).

The concatenation for *Call* is defined by:

$$\begin{aligned} concat'(s \overset{Call}{\rightsquigarrow} v_1, v_1 \overset{(_i}{\rightsquigarrow} v_2) &= s \overset{Call}{\rightsquigarrow} v_2 \\ concat'(s \overset{Call}{\rightsquigarrow} v_1, v_1 \overset{\text{empty},)_j, *}{\rightsquigarrow} v_2) &= \text{NO EDGE!} \end{aligned}$$

The concatenation with $(_i$ results in a *Call* path. The concatenation with the other three edge annotations does not produce an edge—since the *Call* path ends on a sequence of call edges (e.g., $(_i(_j$ or $(_i(_j(_k$, etc.), the indirect flow due to edges with empty, $)_j$, or $*$ annotations is accounted for in *Summarize*.

The concatenation for *nCall* is defined by:

$$\begin{aligned} concat'(s \overset{nCall}{\rightsquigarrow} v_1, v_1 \overset{(_i}{\rightsquigarrow} v_2) &= s \overset{Call}{\rightsquigarrow} v_2 \\ concat'(s \overset{nCall}{\rightsquigarrow} v_1, v_1 \overset{\text{empty},)_j, *}{\rightsquigarrow} v_2) &= s \overset{nCall}{\rightsquigarrow} v_2 \end{aligned}$$

In the first case, the resulting path is a *Call* path, and in the second case, the resulting path is an *nCall* path.

For our example in Figure 3.3, there are the following path edges with source `LOCFLG`:

$$\begin{aligned} &\texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get32.off},\ \texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get16.off}, \\ &\texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get16.i1},\ \texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get16.ret}, \\ &\texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get32.i1},\ \texttt{LOCFLG} \overset{Call}{\rightsquigarrow} \texttt{get32.ret}, \\ &\texttt{LOCFLG} \overset{nCall}{\rightsquigarrow} \texttt{readLOC.i1} \end{aligned}$$

There is a single path, namely a *Call* path, from `LOCFLG` to `get32.ret` and no path edges are added through `get32.ret`. Thus, flow from `LOCFLG` to `readLOC.i2` and subsequently to `ph_long` is, precisely, never discovered. The last edge, namely `LOCFLG` $\overset{nCall}{\rightsquigarrow}$ `readLOC.i1`, results from *Summarize*—there is an edge `LOCFLG` $\rightsquigarrow$ `readLOC.i1` in $\mathcal{FG}^*$ which is converted into the *nCall* path edge.

4.5.4 Termination, Complexity and Correctness

Termination. One can easily see that there are finite number of new edges added in *Summarize*; therefore, $\mathcal{FG}^*$ reaches a fixed point and *Summarize* terminates. Similarly, there are only 2 path edges between s and a node v; therefore $\mathcal{FG}_p$ reaches a fixed point and *Propagate* terminates.

Complexity. Let N be the size of the program being analyzed—that is, the number of statements, the number of methods and the number of variables is of order N. For each pair of nodes v_i, $v_j \in \mathcal{FG}^*$ there could be at most 3 edges between them: (1) a $*$-annotated edge, (2) an empty edge, or (3) *one* of a $(_{...}$ edge or a $)_{...}$ edge. Thus there are at most $O(N^2)$ edges that are processed on the worklist *WL* in *Summarize*. For each edge the algorithm does at most $O(N)$ work examining successor edges (lines 3-5), or examining predecessor edges (lines 6-8). Therefore, the complexity of *Summarize* is $O(N^3)$.

Consider *Propagate* and a given source s. There are only 2 possible path edges between s and a node v, and thus only $O(N)$ paths are processed on the worklist. For each path the algorithm does at most $O(N)$ work examining successor edges (lines 3-4). Thus, *Propagate* does $O(N^2)$ work for a given source s and $O(N^3)$ work for all sources.

Correctness. For correctness, we need to show that any run-time flow path $s \mapsto \ldots o_1.f_1 \mapsto \ldots o_2.f_2 \mapsto \ldots r$ has an appropriate analysis representative in $\mathcal{FG}_p$. The path consists of segments $s \mapsto \ldots o_1.f_1$, $o_1.f_1 \mapsto \ldots o_2.f_2$, etc. where all intermediate nodes between dereferences are local variables that are unique for their creating stack frame.

Consider a segment $s \mapsto \ldots o_1.f_1$. This segment can have the following structure: $s \mapsto fr_1 \overset{ret}{\mapsto} \ldots fr_k \overset{call}{\mapsto} \ldots fr_n \mapsto o_1.f_1$, where each fr_i represents flow within a

method stack frame (i.e., a variable-to-variable flow sequence that starts at variable $v_1 \in fr_i$ and ends at variable $v_2 \in fr_i$). Edge $fr_1 \overset{ret}{\mapsto} fr_2$ denotes that frame fr_1 returns into frame fr_2, and edge $fr_k \overset{call}{\mapsto} fr_{k+1}$ denotes that frame fr_k calls frame fr_{k+1}. Within each frame fr_i there are sequences of balanced frames (i.e., flow starts at fr_i, enters a sequence of stack frames and returns back into fr_i). Then the following statements are true. First, for each sequence of balanced frames $v_1 \overset{call}{\mapsto} ... \overset{ret}{\mapsto} v_2$ in fr_i, there is a representative edge $v_1 \rightsquigarrow v_2$ in $\mathcal{FG}^*$. Second, for each sequence of returns $s \mapsto fr_1 \overset{ret}{\mapsto} ...fr_{k-1} \overset{ret}{\mapsto} v$ there is a representative path edge $s \overset{nCall}{\rightsquigarrow} v$ in $\mathcal{FG}_p$. And third, for each sequence of calls $v \overset{call}{\mapsto} fr_{k+1} \dots fr_n \mapsto o_1.f_1$ there are representative edges $v \overset{*}{\rightsquigarrow} p_1.f_1$ in $\mathcal{FG}_p$ for each variable p_1 s.t. p_1 points to o_1, and field f_1 is read through p_1. The proofs of these statements are outlined in Appendix B. Note that recursion is handled by the analysis and the proofs: any of the sequences (1) $v_1 \overset{call}{\mapsto} ... \overset{ret}{\mapsto} v_2$, (2) $fr_1 \overset{ret}{\mapsto} ...fr_{k-1} \overset{ret}{\mapsto} v$, and (3) $v \overset{call}{\mapsto} fr_{k+1} \dots fr_n \mapsto o_1.f_1$ could be recursive.

Thus, segment $s \mapsto fr_1 \overset{ret}{\mapsto} \dots fr_k \overset{call}{\mapsto} \dots fr_n \mapsto o_1.f_1$ will have analysis representative(s) $s \overset{nCall}{\rightsquigarrow} p_1.f_1$ in $\mathcal{FG}_p$. One can see that the sequence $s \mapsto \dots o_1.f_1 \mapsto \dots o_2.f_2 \mapsto \dots r$ will have analysis representative $s \overset{Call}{\rightsquigarrow} r$ in $\mathcal{FG}_p$ if it ends on a sequence of calls, and $s \overset{nCall}{\rightsquigarrow} r$ otherwise.

4.5.5 Deep Flow Analysis

This section considers *deep flow* analysis. Section 4.5.6 presents the algorithm for deep flow computation for the purpose of detecting confidentiality violations. Section 4.5.7 presents the dual algorithm for deep flow computation for the purpose of detecting integrity violations.

Each algorithm takes as input the summarized flow graph $\mathcal{FG}^*$, a source variable s, which is a sensitive variable or field in *Cls*, and a set of untrusted variables *Sinks*, which contains all placeholder variables `ph_X` from `main`. The output of each algorithm is a boolean result. For confidentiality inference **true** means that there is no information flow, shallow or deep, from sensitive variable s; **false** means that there could be information flow, shallow or deep, from s into some untrusted variable resulting in potential violation of confidentiality. Analogously, for integrity

inference, **true** means that there is no information flow, shallow or deep, into sensitive variable s; **false** means that there could be information flow, shallow or deep, into s from some untrusted variable resulting in potential violation of integrity.

The algorithms use two auxiliary functions, $FShallow(s)$ and $BShallow(s)$, where s is an arbitrary node in the flow graph $\mathcal{FG}^*$. Function $FShallow(s)$ returns the set of nodes reachable forward through shallow flow from s. This set is computed by *Propagate* in Figure 4.18: $FShallow(s) = \{v \mid s \overset{p}{\rightsquigarrow} v \in \mathcal{FG}_p\}$. Function $BShallow(s)$ returns the set of nodes reachable *backwards* through shallow flow from s—that is, nodes v such that there is shallow flow from v to s. This set is computed on demand analogously to *Propagate*, only backwards. The computation keeps path edges e_1: $v_1 \overset{p}{\rightsquigarrow} s$ on the worklist, and examines edges e_2: $v_2 \overset{a}{\rightsquigarrow} v_1$ from $\mathcal{FG}^*$; e_2 is concatenated with e_1 and the resulting edge e_3 (if any) is added to the backward path graph $\mathcal{FG}^{-1}$ and to the worklist. Note that one needs different path annotations for backward reachability. Again, there are two kinds of path annotations: $Call^{-1}$ and $nCall^{-1}$. $Call^{-1}$ denotes flow paths that begin with a call sequence. $nCall^{-1}$ denotes paths that do not begin with a call sequence. These paths could be: (1) empty paths, (2) paths that begin with a return sequence, or (3) paths that begin with $*$. Analogously to forward propagation, we need to consider concatenation of each possible edge annotation (i.e., (1) empty, (2) $(_i$, (3) $)_j$, and (4) $*$ with each possible path annotation (i.e., (1) $Call^{-1}$, and (2) $nCall^{-1}$). The concatenation for $Call^{-1}$ is defined by:

$$\begin{aligned} concat'(v_2 \overset{(_i}{\rightsquigarrow} v_1, v_1 \overset{Call^{-1}}{\rightsquigarrow} s) &= v_2 \overset{Call^{-1}}{\rightsquigarrow} s \\ concat'(v_2 \overset{\text{empty},\ )_j,\ *}{\rightsquigarrow} v_1, v_1 \overset{Call^{-1}}{\rightsquigarrow} s) &= v_2 \overset{nCall^{-1}}{\rightsquigarrow} s \end{aligned}$$

The concatenation for $nCall^{-1}$ is defined by:

$$\begin{aligned} concat'(v_2 \overset{(_i}{\rightsquigarrow} v_1, v_1 \overset{nCall^{-1}}{\rightsquigarrow} s) &= \text{NO EDGE!} \\ concat'(v_2 \overset{\text{empty},\ )_j,\ *}{\rightsquigarrow} v_1, v_1 \overset{nCall^{-1}}{\rightsquigarrow} s) &= v_2 \overset{nCall^{-1}}{\rightsquigarrow} s \end{aligned}$$

The union of $FShallow(s)$ and $BShallow(s)$ gives the set of variables that contain shallow information (via shallow flows) of variable s.

procedure **DeepPropagate**
input $\mathcal{FG}^*$: summarized flow graph
s: source node *Sinks*: untrusted nodes
output *result*: boolean
initialize $SWL = \{s\}$
[1] while $SWL \neq \emptyset$ do
[2] remove s from SWL
[3] if $FShallow(s) \cap Sinks \neq \emptyset$ **return false;**
[4] if s is of reference type
[5] foreach $v \in \{s\} \cup FShallow(s) \cup BShallow(s)$ do
[6] foreach indirect read $s' = v.f$ do
[7] add s' to SWL
[8] **return true;**

Figure 4.19: Computation of deep flow.

4.5.6 Confidentiality Inference

Recall from Section 3.4 that if a sensitive variable s is a reference variable, there may be flow from the object structure rooted at s. By definition (Section 3.4), deep flow from s into r occurs if there is a statement $s' = v.f$ such that v points to some object reachable on a sequence of field dereferences from s, and there is shallow flow from s' to r.

Procedure *DeepPropagate* in Figure 4.19 states the algorithm for confidentiality inference. The algorithm uses a worklist of sources, SWL, which is initialized with s. It finds the set of variables r such that there is shallow flow from s to r, and checks if any of these variables is untrusted (line 3). Lines 4-7 are necessary for deep flow computation. Lines 5-6 examine each valid alias v of s and check if there is a field read from v; if there is such a read statement, the left-hand side s' of the statement is added to SWL.

Example. Recall the example of deep flow from Figures 3.3 and 4.15. Source node s is `tmpbuf` and *Sinks* includes all placeholder variables from `main` in Figure 4.15. Initially `tmpbuf` is added to SWL. Then `tmpbuf` is taken off the worklist and we have $FShallow(\texttt{tmpbuf}) = \{\texttt{get32.b}, \texttt{get16.b}\}$. This set does not intersect with *Sinks* and the analysis proceeds. Set $BShallow(\texttt{tmpbuf})$ is empty and lines 5-6 in the algorithm examine only `tmpbuf`, `get32.b`, and `get16.b` for indirect reads.

There is indirect read from `get16.b` on line 11 in Figure 3.3 and `get16.b1` is added to *SWL*. The algorithm proceeds to compute *FShallow*(`get16.b1`) which includes `ph_long` from `main`; therefore, the result is **false**.

It remains to show that this algorithm actually computes deep flow as defined in Section 3.4 and reiterated in the beginning of this section. Intuitively, one needs to show that every relevant field read $s' = v.f$ is examined by our algorithm. This is done by considering induction on the length of the field path from s. Assume that each field read $s' = v.f_k$ such that v is aliased with $s.f_1...f_{k-1}$ is examined (recall that s is the source sensitive variable). We need to show that every field read $s'' = v'.f_{k+1}$ such that v' is aliased with $s.f_1...f_{k-1}.f_k$, is examined as well. Our analysis assumes that each object field is read at least once. That is, there is at least one read statement $s' = v.f_k$ which reads field f_k of object $s.f_1...f_{k-1}$. By the inductive hypothesis, statement $s' = v.f_k$ is examined and s' is processed on the worklist and sets *FShallow*(s') and *BSshallow*(s') are computed. Recall statement $s'' = v'.f_{k+1}$. Clearly, v' and s' are aliased since they refer to the same object, namely the one referred to by $s.f_1...f_{k-1}.f_k$. Thus, v' must be included in one of these two sets (either there is shallow flow from s' to v' and $v' \in$ *FShallow*(s') or there is shallow flow from v' to s' and $v' \in$ *BShallow*(s')). Therefore, statement $s'' = v'.f_{k+1}$ is examined as well.

4.5.7 Integrity Inference

The definition of deep flow for the purpose of detecting integrity is the following. There is flow from variable r into some sensitive variable s if there is a statement $v.f = s'$ such that there is shallow flow from r to s' and v points to some object reachable on a sequence of field dereferences from s.

The integrity inference is the dual of the confidentiality inference in Figure 4.19. It has the same inputs as the algorithm for confidentiality inference and differs only slightly (the two adjustments are highlighted in bold):

[3] if **BShallow**(**s**) $\cap$ *Sinks* $\neq \emptyset$ return false

[4] if s is of reference type

[5] foreach $v \in \{s\} \cup$ *FShallow*(s) $\cup$ *BShallow*(s) do

[6] foreach indirect **write v.f = s′** do

[7] add s' to SWL

Line 3 considers set $BShallow(s)$—that is, the set of variables r such that there is shallow flow from r into s. If some of these variables is in *Sinks* the algorithm returns false. Lines 5-6 find all valid aliases v of s and examine field writes $v.f = s'$. Each s' is part of the object structure rooted at s; it is put on the worklist and subsequently, line 3 checks for violating shallow flow into s'.

Note that the described information flow analysis is again a whole-program analysis. As with the points-to analysis the placeholder `main` enables the use of the whole-program information flow analysis. The `main` method approximates all possible clients that could be built on top of *Cls* and thus the result of the whole-program information flow analysis includes all flows that could result from individual clients (clearly, under the constraints in Section 2.3.2).

CHAPTER 5
Empirical Results

The static analysis framework is implemented in Java using Soot 2.2.3 [18] and Spark [13]. It uses the Andersen-style points-to analysis provided by Spark. We performed the analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a 900MHz Sun Fire 380R machine with 4GB of RAM. The implementation, which includes Soot and Spark, was run with a maximum heap size of 800MB.

Native methods are handled using the models provided by Soot. Reflection is handled by specifying the dynamically loaded classes which Spark uses to appropriately resolve reflection calls. This approach is used in other whole-program analyses based on Soot and Spark [19].

The empirical study addresses two important issues.

First, it addresses the issue of *analysis precision*—that is, how often the analyses report safe fields, methods and parameters as unsafe (e.g., how often the ownership analysis reports confined fields as exposed; or how often the immutability analysis reports an immutable parameter as mutable). Precision is crucial for static analyses developed for inferring the implemented security-related properties: imprecise analysis is not merely useless, but also confusing, and may discourage developers from using the tool. For example, an imprecise information flow analysis may output a class diagram without the `safe` constraint on a field f (i.e., warning that f is tampered), while in fact f is safe. Developers could spend valuable time examining potentially large amounts of code until they determine that the warning is due to analysis imprecision and not to insecure information flow. Recall that the analyses are conservative—that is, if a field is reported as `owned`, `read-only`, `confidential` or `safe`, then it is in fact `owned`, `read-only`, `confidential` or `safe`.

Second, the study addresses the issue of *analysis scalability*—that is, do the analyses have acceptable cost? Analysis scalability is important as well—if the analysis runs in hours or days, developers would be less likely to seek the benefits of the tool.

5.1 Results on Software Components

(1)Component	(2)Functionality	(3)#Classes / /#Functionality	(4)#Fields	(5)#Reachable Methods
`gzip`	GZIP IO streams	199/6	16	3481
`zip`	ZIP IO streams	194/6	73	3506
`checked`	IO stream checksums	189/4	3	3428
`collator`	text collation	203/15	148	3535
`breaks`	text break	193/13	117	3487
`number`	number formatting	198/10	14	3541

Table 5.1: Information of Java components.

We evaluated the framework and the analyses on several Java components from the packages `java.text` and `java.util.zip` (these components were used in related analyses [8] and [9]).[12] The components are described in the first three columns of Table 5.1. Each component contains the set of classes in *Cls* (i.e., the functionality classes which are defined in the components and provide component functionality, plus all other classes that are directly or transitively referenced); the total number of classes and the number of functionality classes is shown in column (3). The number of fields in functionality classes is shown in column (4). The last column shows the number of methods in all classes (i.e., functionality classes and library classes), determined to be reachable by Spark.

5.1.1 Analysis Precision

In this section we report our results on the inference of ownership, immutability (field, parameter and method), and information flow (confidentiality and integrity), and address the issue of precision.

We applied the **ownership analysis** described in Section 4.3 to instance fields in functionality classes. The results are reported in Table 5.2.[13] We applied the **immutability analyses** described in Section 4.4, on instance fields in functionality classes, on methods in functionality classes, and on parameters of methods in functionality classes. The results are reported in Table 5.3. Finally, we applied the

[12]The current paper does not include one of the 7 components used in previous work, namely `date`. We were unable to run this component with our current Soot infrastructure, because Soot throws exception and terminates on this component.

[13]Results are obtained with a slightly different version of ownership algorithm.

Program	#Instance Fields (reference type)	#Owned Fields
gzip	7	4(57%)
zip	10	5(50%)
checked	2	0(0%)
collator	17	9(53%)
breaks	7	0(0%)
number	3	1 (33%)

Table 5.2: Owned fields.

Program	#Fields (reference)	#Immut-able	#Methods	#Immut-able	#Para-meters	#Immut-able
gzip	7	1 (14.29%)	25	1(4%)	33	3(9%)
zip	10	0 (0.00%)	48	11(23%)	60	16(27%)
checked	2	2 (100%)	11	5(45%)	14	7(50%)
collator	17	5 (29.41%)	80	51(64%)	100	63(63%)
breaks	7	6 (85.71%)	56	36(64%)	55	37(67%)
number	3	0 (0.0%)	81	42(52%)	100	47(47%)

Table 5.3: Immutable fields, methods and parameters.

Program	#Fields (non-public)	#Leaked (shallow)	#Leaked (all)	#Tampered (shallow)	#Tampered (all)
gzip	15	2(13.33%)	2(13.33%)	5(33%)	5(33%)
zip	29	9(31.03%)	13(44.83%)	16(55%)	18(62%)
checked	3	3(100%)	3(100%)	2(67%)	2(67%)
collator	134	22(16.42%)	33(24.63%)	11(8%)	16(12%)
breaks	241	6 (2.49%)	7 (2.90%)	5(2%)	5(2%)
number	66	22 (33.3%)	25 (37.88%)	6(9%)	6(9%)

Table 5.4: Confidentiality (fields leaked to client code) and integrity (fields tampered by client code).

information flow analyses described in Section 4.5 on sensitive fields (i.e., non-public fields) in functionality classes. The results from confidentiality and integrity inference are shown in Table 5.4.

For each of the three analyses we examined manually the reported results. We examined each non-owned field, mutable field/parameter/method, and leaked or tampered field, and attempted to construct a client that would expose appropriate non-ownership, mutability or information flow. In all cases, we were able to construct such a client. Thus the analysis is precise—fields reported as non-owned are indeed non-owned; fields, parameters and methods reported as mutable are indeed mutable;

and fields reported as leaked or tampered are indeed leaked or tampered.

5.1.2 Analysis Cost

In this section we report on the cost of the analyses. The cost of the three analyses is given in Table 5.5. The ownership analysis typically runs within 20 seconds. The immutability analysis, which includes the analysis of fields, methods and parameters, runs within seconds as well. Finally, the information flow analysis, which includes both confidentiality and integrity inference, runs in about 10 seconds.

Program	Ownership	Immutability	Information Flow
`gzip`	19s	24s	6s
`zip`	19s	42s	7s
`checked`	18s	20s	5s
`collator`	20s	40s	11s
`breaks`	22s	18s	8s
`number`	29s	29s	9s

Table 5.5: Analysis time.

Our empirical results indicate that the analysis framework and the specific analyses are precise and practical. Although the software components that we reported on are relatively small, our results so far indicate that the analyses work precisely and efficiently on substantially larger programs as well.[14] The analysis is the first and arguably most important step towards practical model-based reasoning about ownership, immutability, information flow and other security properties. Although more work remains to be done, we believe that our analysis framework presents a promising foundation.

5.2 Results on Complete Programs

Our benchmark suite includes several relatively small applications, `soot-c` and `sablecc-j` from the Ashes suite [20], relatively large benchmarks from the DaCapo benchmark suite version beta051009 [21] and the Polyglot Java front-end. The suite is described in Table 5.6. The number of user classes and user methods fetched by

[14] Immutability Analysis and information flow analysis are cubic time analysis, the same as the underlying Andersen's points-to analysis. Ownership analysis has complexity of $O(N^4)$.

(1)Program	(2)Description	(3)Size		
		User Classes	User Methods	#Reachable Methods
jdepend-2.9.1	A metrics suite for Java	17	225	3962
javad	Classfile decompiler	41	156	3838
JATLite-0.4	Java Agent Template	45	442	6279
undo	Undoable e-mail Store	237	1709	5644
soot-c	Analysis framework for Java	579	2935	6046
sablecc-j	Java parser generator	300	2024	7970
polyglot-1.3.2	Extensible Java compiler	267	3418	7449
antlr	Language recognition tool	126	1738	5102
bloat	Java bytecode optimizer	289	3232	6402
jython	Python interpreter	163	2892	5606
pmd	Java source code analyzer	718	7057	8653
ps	Postscript interpreter	200	908	5396

Table 5.6: Information about the Java benchmarks.

Soot are shown in the first two columns of multicolumn (3); these numbers exclude the standard libraries but include other libraries shipped with the application. The last column shows the number of methods (user and library), determined to be reachable by Spark.

Program	#Instance Fields (reference type)	#Owned Fields
jdepend	33	19 (58%)
javad	40	19 (48%)
JATLite	142	35 (27%)
undo	325	63 (19%)
soot	340	62 (18%)
sablecc	304	30 (10%)
polyglot	435	57 (13%)
antlr	161	27 (17%)
bloat	529	81 (15%)
jython	215	45 (21%)
pmd	125	40 (32%)
ps	19	7 (37%)
Average		26%

Table 5.7: Owned fields.

We applied the **ownership analysis** described in Section 4.3 to non-static instance fields in non-library classes. The results are reported in Table 5.7. We

Program	Fields	Immut-able	Methods	Immut-able	Param-eters	Immut-able
jdepend	33	6 (18%)	30	7 (23.3%)	62	31 (50%)
javad	40	40 (100%)	13	1 (7.7%)	26	7 (26.9%)
JATLite	142	13 (9.2%)	85	11 (12.9%)	197	53 (26.9%)
undo	325	162 (50%)	135	31 (23.0%)	317	131 (41.3%)
soot	340	57 (16.8%)	369	213 (57.7%)	843	533 (63.2%)
sablecc	304	40 (13.2%)	784	333 (42.5%)	1577	850 (53.9%)
polyglot	435	92 (21.1%)	843	212 (25.1%)	2180	873 (40.0%)
antlr	161	25 (15.5%)	276	49 (17.8%)	635	231 (36.4%)
bloat	529	73 (13.8%)	1061	107 (10.1%)	2145	401 (18.7%)
jython	215	21 (9.8%)	529	140 (26.5%)	1194	448(37.5%)
pmd	125	42 (33.6%)	406	129 (31.8%)	1103	365 (33.1%)
ps	19	8 (42.1%)	193	172 (89.1%)	388	357 (92.0%)

Table 5.8: Immutable fields, methods and parameters.

Program	#Fields	#Leaked (shallow)	#Leaked (all)	#Tampered (shallow)	#Tampered (all)
jdepend	53	12(23%)	23(43%)	20(38%)	31(58%)
javad	73	7(10%)	38(52%)	40(55%)	64(88%)
JATLite	180	80(44%)	114(63%)	0(0%)	2(1%)
undo	605	172(28%)	288(48%)	29(5%)	102(17%)
soot	435	83(19%)	285(66%)	38(9%)	145(33%)
sablecc	365	50(14%)	251(69%)	12(3%)	20(5%)
polyglot	253	3(1%)	4(2%)	1(.4%)	1(.4%)
antlr	433	126(30%)	201(46%)	0(0%)	20(5%)
bloat	790	296(37%)	437(55%)	101(13%)	347(44%)
jython	279	81(29%)	111(40%)	9(3%)	42(15%)
pmd	1826	23(1%)	50(3%)	3(.2%)	5(.3%)
ps	28	5(18%)	8(29%)	3(11%)	4(14%)

Table 5.9: Columns 3 and 4: Confidentiality; Columns 5 and 6: Integrity.

applied the **immutability analyses** described in Section 4.4, on non-static instance fields in non-library classes, public methods in public classes, and parameters of those methods. The results are reported in Table 5.8. Finally, we applied the **information flow analyses** described in Section 4.5 on sensitive fields (i.e., non-public non-static fields) in functionality classes. The results from confidentiality and integrity inference are shown in Table 5.9.

On average, the ownership analysis identified 26% of the fields as owned (column #Owned). Also, on average, the immutability analysis identified 29% of the

fields as read-only (column #Immutable). And 43% of the examined data were determined to be `deep leaked`, 23% were determined to be `deep tampered`.

5.2.1 Analysis Precision

The issue of analysis precision is of crucial importance for software tools. If the ownership analysis is imprecise, it may report that an association is non-owned while in reality it is owned (i.e., the analysis reports that certain representation may be exposed while in fact it is not). Similarly, the immutability analysis may report that an association is non-read-only, while in reality it is. Such information is not useful and may confuse the user. For example, if a user attempts to verify lack of representation exposure, imprecision will mean that potentially large amount of code will have to be examined manually. Therefore, imprecision must be carefully evaluated by analysis designers.

For each of the three analyses we examined manually the reported results. We performed a study of absolute precision [7, 9] on a subset of the fields. Specifically, we considered all tracked fields in the two smallest benchmarks, `jdepend` and `javad`, and all fields in the class with the largest number of tracked fields for the three largest benchmarks, `polyglot, sablecc` and `pmd` (the size metric that we used was the number of reachable methods, shown in Column (3) of Figure 5.6). We examined each non-owned field, mutable field/parameter/method, and leaked/tampered field, and attempted to find an execution path that would expose appropriate non-ownership, mutability or information flow. Of 88 reported as non-owned fields, we were able to show exposure; that is, for this set of fields the ownership analysis achieved perfect precision. Similarly, of 76 non-readonly fields, in all but 3 cases we were able to show mutability. We examined manually 69 fields that are reported as leaked (column (4) in Table 5.9), and in all but 1 case we were able to show the field leakage. Furthermore, we examined 95 fields that are reported as being tampered, and in all cases we successfully found execution paths that tampered the field. Thus the analysis achieved very good precision as well.

In short, the majority of imprecisions are mostly due to context-insensitive object naming in the underlying points-to analysis. However, using context-sensitive

object naming, which is expensive, may not be justified—it will likely improve precision only marginally over the current practical analysis which is already adequately precise.

5.2.2 Analysis Cost

In this section we report on the cost of the analyses. The cost of the three analyses is given in Table 5.10. The ownership analysis typically runs within 12 minutes, except for 1 case where it runs about 42 minutes due to the complex object graph. The immutability analysis, which includes the analysis of fields, methods and parameters, runs within 5 minutes for most large applications, except for 2 cases where it runs about 9.5 minutes and another where it runs about 14.5 minutes, due to the large number of parameters (over 2100). Finally, the information flow analysis, which includes both confidentiality and integrity inference, runs within 8 minutes in all cases.

Program	Points-to Analysis	Ownership Analysis	Immutability Analysis	Information Flow Analysis
`jdepend`	1m35s	28s	20s	12s
`javad`	1m33s	18s	4s	9s
`JATLite`	2m37s	2m14s	27s	59s
`undo`	3m3s	6m53s	1m13s	46s
`soot`	2m23s	12m	1m59s	1m31s
`sablecc`	3m5s	3m40s	4m30s	1m39s
`polyglot`	9m39s	42m5s	9m22s	7s
`antlr`	2m25s	2m19s	68s	1m12s
`bloat`	2m36s	5m19s	14m21s	7m38s
`jython`	1m58s	8m36s	4m21s	3m48s
`pmd`	4m17s	5m54s	3m41s	47s
`ps`	2m19s	8m38s	29s	27s

Table 5.10: Analysis time.

Figure 5.1 shows the analysis times versus program size. Each chart represents for each analysis, illustrating how the analysis time differs when the program size (measured with reachable methods) increases. In each chart, the X axis shows the number of analyzed reachable methods, in increasing order, the Y axis shows the represented analysis time. Generally, the analysis time increases when program size

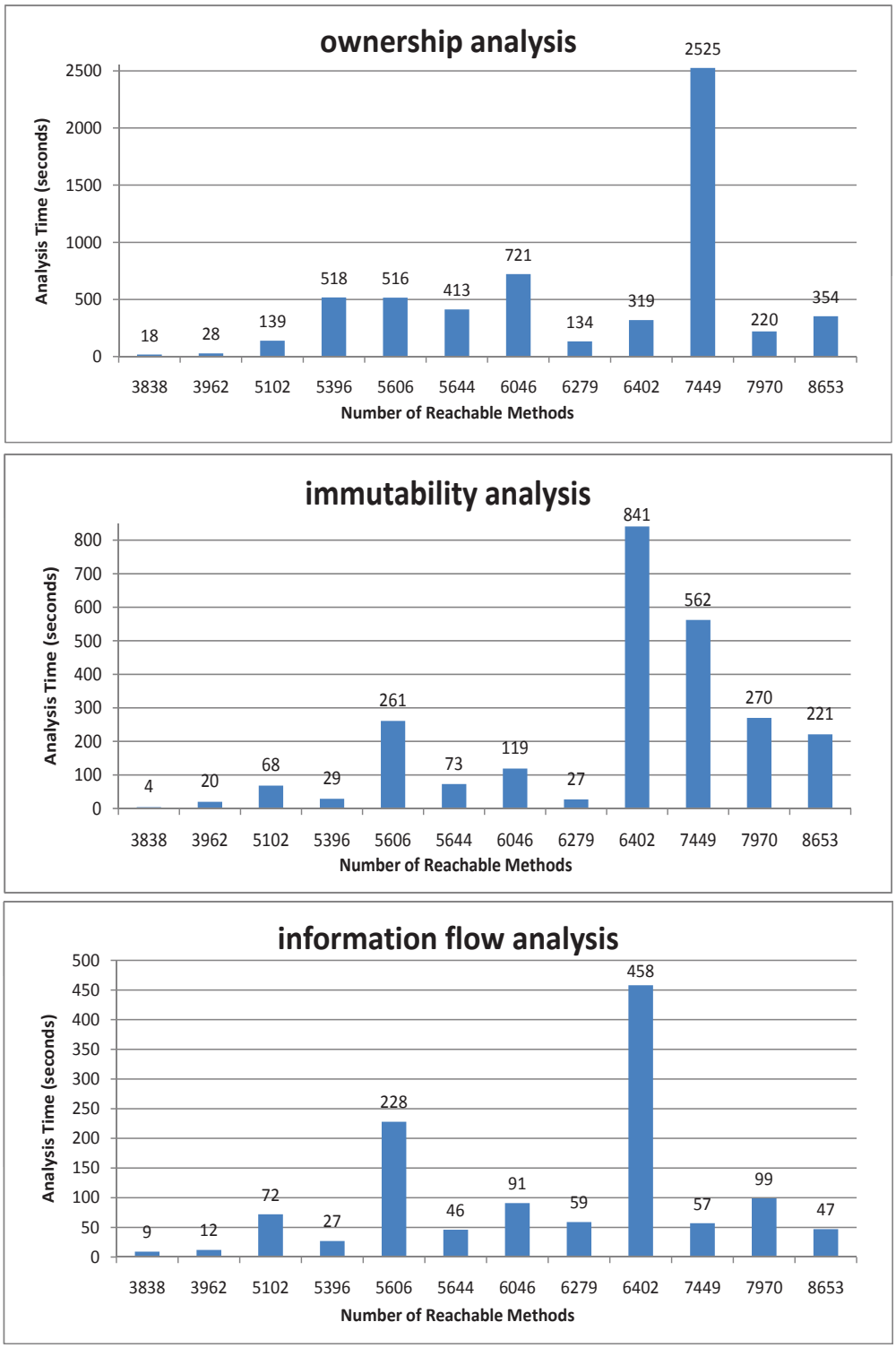

Figure 5.1: Analysis Time versus Program Size.

increases. However, there exists peaks for the analysis time, because the complexity of the analysis not only depends on the program size, but also on the complexity of the object graph (how the objects access each other).

5.3 Conclusions

The empirical study leads to the following observations. First, the analyses scale to large programs, analyzing close to ten thousand reachable methods mostly within 15 minutes. Second, the ownership and immutability models capture well the meaning of these notions in modeling. Clarke et al. [2] argue that the owners-as-dominators model captures well the notions of ownership and composition in modeling; our study reaffirmed this observation. The immutability model captures relationships intuitively as well. It also led us to a bug in the example code of Appendix A. The information flow model captures well the notion of data confidentiality and data integrity. Overall, the analyses produce useful results, which are easy to interpret in the context of UML class diagrams. Third, the analyses are relatively precise, rarely missing `owned`, `read-only`, `confidential` and `safe` properties. In summary, the empirical study indicates that the analyses can effectively support model-driven development and reasoning about software quality and security.

CHAPTER 6
Applications of the Framework

Our static analysis framework could infer implemented security-related properties. Within this framework we infer ownership, immutability and information flow for the protection of object access, data confidentiality and integrity. These inferences reveal information about object access and information flow in the program, and may help uncover serious vulnerabilities. In this chapter, several applications of the framework are shown to illustrate how the framework could help in program understanding and other domains.

6.1 An Application of Ownership Analysis

Concurrent programming with shared memory is notoriously difficult. Programs often exhibit incorrect behavior because of unintended interaction between threads; concurrency bugs such as data races are difficult to find.

In recent years concurrent programming with shared memory has received considerable attention due to the advances in multi-core processors. The vast majority of work falls into two categories (1) work on error prevention, and (2) work on error detection. Work on error prevention focuses on language-based techniques (i.e., type systems) which disallow concurrency errors such as data races and deadlocks. Work on error detection includes static and (mostly) dynamic analysis techniques that detect errors such as data races, deadlocks and atomicity violations.

Surprisingly, little has been done on understanding concurrent programs, or more specifically on understanding the structure of sharing in concurrent programs. Understanding the structure of sharing in real-world concurrent programs has important benefits. First, it can help developers, testers and maintainers uncover and prevent concurrency errors. Second, it can help work on language-based error prevention by guiding language designers towards practical type systems. Third, it can lead to more efficient algorithms for error detection.

We consider the problem where the target program is an *open, concurrent*

Java program — that is, a library of Java classes designed for use by multi-threaded clients. We propose three structural patterns for object sharing: we classify shared objects as *central*, *owned* or *distributed*. Informally, a *central* object is an object *directly* accessed by client threads (e.g., method `run()` executed on a client thread, calls a method on the central object). An *owned* object is an object *indirectly* accessed by client threads (e.g, `run()` executes a method m on a central object, and m in turn executes a method on the owned object); however, an owned object is "dominated" by an "owner" object, meaning (informally) that each access to that object goes through the "owner" object. A *distributed* object is indirectly accessed by client threads as well; however, a distributed object is not dominated by an "owner" object; instead, access is distributed through multiple objects. We conjecture that well-structured concurrent programs should emphasize the role of ownership — they should strive for a maximal number of owned objects and for a minimal number of distributed objects.

We argue that these patterns facilitate the understanding of concurrent programs. We conjecture that well-structured concurrent programs should emphasize the role of ownership — they should strive for a larger number of owned objects and for a minimal number of distributed objects.

6.1.1 Problem Setting

We consider a set *Cls* of interacting classes that constitute the open concurrent program. The classes in *Cls* are designed for use by clients that spawn multiple threads; as a result, the classes are intended to be "thread-safe" and typically include multiple synchronized methods. In addition, we consider a set *Int* of methods and fields from classes in *Cls*. These methods and fields define the interface to the client. In general, *Int* could contain a small subset of all fields and methods from *Cls* or it may contain the entire set of public methods and fields in *Cls*. For our purposes, *Int* contains the entire set of public methods and fields in *Cls*.

A client of *Cls* is any arbitrary Java class that calls methods from *Int* and reads/writes fields from *Int*, and does not access any methods/fields from *Cls* that are not in *Int*. We denote by *AllClients(Int)* the set of all possible clients for *Int*;

```
public class BaseRecordManager {
   private RecordFile _file;
   private PhysicalRowIdManager _physMgr;
   BaseRecordManager(RecordFile file) {
1     _file = new RecordFile();
2     _physMgr = new PhysicalRowIdManager(_file);
   }
   public synchronized TransactionManager getTxnManager() {
      return _file.txnMgr;
   }
   public synchronized void close() {
      _file.close();
      _physMgr.close();
   }
}
public final class RecordFile {
   final TransactionManager txnMgr;
   RecordFile() {
3     txnMgr = new TransactionManager(this);
   }
   ...
}
public class TransactionManager {
   private RecordFile owner;
   TransactionManger(RecordFile file) {
      owner = file;
   }
   public void synchronizeLog() {
      ...
      owner.synch();
   }
}
```

Figure 6.1: `BaseRecordManager` **from** `jdmb`**.**

clearly, this set is infinite. As described in Section 2.3.2, we assume that *Cls* is closed with respect to *Int*: for any arbitrary client $C \in$ *AllSuites(Int)*, any class that could be referenced during the execution of C is included in *Cls*. In other words, we assume clients of the open concurrent program that only exercise interactions among classes from the given set *Cls* (i.e., the current concurrent program).

We illustrate the problem setting with an example taken from one of our bench-

```
class PlaceholderClient {
   public static void main() {
4     BaseRecordManager brm = new BaseRecordManager();
5     TransactionManager tm = brm.getTxnManager();
      brm.close();
      tm.synchronizeLog();
   }
}
```

Figure 6.2: Client of `BaseRecordManager`.

mark programs, `jdbm`, an open concurrent program which impelements a transactional persistence engine. Figure 6.1 shows excerpts from classes `BaseRecordManager`, `RecordFile` and `TransactionManager`; these classes and their methods are public. *Int* consists of the public methods in classes `BaseRecordManager`, `TransactionManager` and `RecordFile`. An arbitrary client that meets the constraints outlined above is shown in Figure 6.2.

Note that although the clients in *AllClients*(*Int*) are single-threaded (i.e., a client is a single method `main`), these clients are sufficient to enable reasoning about the interactions of actually multithreaded clients and *Cls*. Clearly, every object accessed in the client's `main` method is an object that can be directly shared by multiple threads: every method/field accessed in `main` can be directly called/accessed by multiple threads.

6.1.2 Run-time Method Sequence Graph

A *run-time object graph* Og^r, as described in Section 3.2, represents a view of a program execution, while a *run-time method sequence graph* represents another view of a program execution. The nodes in the run-time method sequence graph are the run-time tuples $o^r.m()$, where $o^r.m()$ denotes that instance method m is invoked on receiver o^r. The edges represent the calling relationships between these run-time tuples.

For convenience, we denote field accesses not through `this` (i.e, $p = q.f$, $q \neq$ `this` and $p.f = q$, $p \neq$ `this`), and array accesses (i.e., $p = q[i]$ and $p[i] = q$) as special method calls. Notation $o_2^r.rd$ denotes the execution of a read $p = q.f$

($p \neq$ `this`), where q refers to o_2^r. Similarly, $o_2^r.wr$ denotes the execution a write $p.f = q, p \neq$ `this` where p refers to o_2^r.

Let $C \in AllClients(Int)$ be a client of Cls, and let E_C be an execution of this client. Let $Og_{E_C}^{r+}$ be the run-time method sequence graph for this execution. $Og_{E_C}^{r+}$ is constructed as follows:

- There is an edge $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$ if at some point of the execution E_C, method m_1 invoked on receiver o_1^r executes a call $p.m_2()$ in m_1, where p refers to o_2^r, and the call leads to the invocation of method m_2 on receiver o_2^r.

- There is an edge $o_1^r.m_1'() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$ if at some point of the execution E_C, method m_1' invoked on receiver o_1^r has executed a call `this`$.m_1()$ in m_1', and the call has resulted in edge $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$.

The method sequence graph shows the calling relationships between run-time tuples $o^r.m()$. The above bullets present two cases. In the first case (the first bullet above), method m_1 executing on receiver o_1^r directly calls m_2 on receiver o_2^r through call site $p.m_2()$ which directly results in the appropriate edge $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$. In the second case (the second bullet above), method m_1 "jumps through" calls trough `this` until it reaches a call not through `this`: for example, if there is a method m_1' invoked on receiver o_1^r and m_1' executes a call `this`$.m_1()$, then in turn m_1 executes a call $p.m_2()$, $p \neq$ `this`, where p refers to o_2^r, then there is an edge $o_1^r.m_1'() \rightarrow o_2^r.m_2()$ in the method sequence graph. The method sequence graph reflects transfer of control between distinct objects.

Throughout the paper we use the standard notation for reachability over $Og_{E_C}^{r+}$: $o_1^r.m_1() \rightarrow^* o_2^r.m_2() \in Og_{E_C}^{r+}$ denotes that tuple $o_2^r.m_2()$ is reachable on zero or more edges from tuple $o_1^r.m_1()$. Similarly, $o_1^r.m_1() \rightarrow^+ o_2^r.m_2() \in Og_{E_C}^{r+}$ denotes that tuple $o_2^r.m_2()$ is reachable on one or more edges from tuple $o_1^r.m_1()$.

The object graph and the method sequence graph represent two different views of a program execution. The object graph represents a structural (static) view of the execution — it shows the access relationships between run-time objects and can be used to reason about structural (static) properties such as ownership. On the

other hand, the method sequence graph represents a dynamic view of the execution — it shows the transfer of control between distinct run-time objects.

6.1.3 Defining Patterns of Object Sharing

In this section, we use the object graph and the method sequence graph to define the structural patterns of sharing. Our analysis, presented later, infers static approximations of these graphs, and proceeds to reason about the structure of sharing using these static approximations.

6.1.3.1 Patterns of Object Sharing

Let $C \in \mathit{AllClients}(\mathit{Int})$ be a client of *Cls*, and let E_C be an execution of this client. Let $Og^r_{E_C}$ be the run-time object graph for this execution, and $Og^{r+}_{E_C}$ be the run-time method sequence graph for this execution. The `main` method is treated as a special instance method executed on a special receiver object `root`. Below we define the notion of *potentially shared* objects, and proceed to classify the *potentially shared* objects into *central*, *owned* and *distributed*.

The set of *potentially shared* objects in E_C, denoted by $\mathcal{S}_{E_C}$ is the union of the set of *directly shared* objects denoted by $\mathcal{DS}_{E_C}$, and the set of *indirectly shared* objects denoted by $\mathcal{IS}_{E_C}$.

The set of directly shared objects is defined as follows:

$$\mathcal{DS}_{E_C} = \{o^r \mid \exists \texttt{root.main}() \rightarrow o^r.m() \in Og^{r+}_{E_C}\}.$$

The definition states that there exists an edge in $Og^{r+}_{E_C}$ from `root.main`() to a tuple $o^r.m()$ (i.e., `main` directly calls a method on receiver o^r). The objects o^r are the objects potentially directly accessed by multiple threads — o^r's methods/fields can be directly called/accessed by multiple threads.

The set of indirectly shared objects is defined as follows:

$$\begin{aligned}\mathcal{IS}_{E_C} = \{o^r \mid o^r \notin \mathcal{DS}_{E_C} \wedge \exists o^r_j \in \mathcal{DS}_{E_C} \text{ s.t. } o^r_j \xrightarrow{f}{}^{+} o^r \in Og^r_{E_C} \\ \wedge \exists \texttt{root.main}() \rightarrow^+ o^r.m() \in Og^{r+}_{E_C}\}.\end{aligned}$$

The definition states that an object o^r is indirectly shared, if the following three conditions are met. The first condition, $o^r \notin \mathcal{DS}_{E_C}$, states that o^r must not be a directly shared object. The second condition, $\exists o_j^r \in \mathcal{DS}_{E_C}$ s.t. $o_j^r \xrightarrow{f}{}^{+} o^r \in Og_{E_C}^r$, states that o^r must be a transitively reachable field of some directly shared object o_j^r. This condition excludes "temporary" objects from consideration: if an object o_1^r is not reachable from a directly shared object on a sequence of field accesses, then this means that o_1^r is local to the execution of a particular method in a thread, and o_1^r cannot be accessed simultaneously by multiple threads. The third condition, $\exists \texttt{root.main}() \rightarrow^+ o^r.m() \in Og_{E_C}^{r+}$, states that there must exist a path in $Og_{E_C}^{r+}$ from `root.main`() to $o^r.m()$ (i.e., the execution of the client's `main` leads to a non-trivial access of object o_r^r; however, this access happens indirectly, that is, through one or more intermediate objects).

Therefore, the set of potentially shared object is as follows:

$$\mathcal{S}_{E_C} = \mathcal{DS}_{E_C} \cup \mathcal{IS}_{E_C}.$$

We are ready to define the set of *central* objects, *owned* objects and *distributed* objects in E_C.

The set of *central* objects in E_C, which we denote by $\mathcal{C}_{E_C}$, is the set of directly shared objects:

$$\mathcal{C}_{E_C} = \mathcal{DS}_{E_C}.$$

The set of *owned* objects in E_C, denoted by $\mathcal{O}_{E_C}$ is defined as follows:

$$\begin{aligned}\mathcal{O}_{E_C} = \{o^r \mid\ & o^r \in \mathcal{IS}_{E_C} \wedge \\ & \exists o_1^r \text{ s.t. } \forall \texttt{root.main}() \rightarrow^+ o^r.m() \in Og_{E_C}^{r+}, \text{ it denotes} \\ & \qquad \texttt{root.main}() \rightarrow^+ o_1.m_1() \rightarrow^+ o^r.m()\}.\end{aligned}$$

The definition states that an object $o^r \in S_{E_C}$ is *owned*, if the following two conditions are met. The first condition, $o^r \in \mathcal{IS}_{E_C}$, states that o^r must be an indirectly shared object. The second condition states that there must exist an object o_1^r, such that all accesses from `root.main`() to o^r, go through object o_1^r. In other words, an owned object is potentially indirectly accessed by multiple client threads; however, these

accesses always go through the same "owner" object.

Finally, the set of *distributed* objects in E_C, which we denote by $\mathcal{D}_{E_C}$ is defined as follows:

$$\mathcal{D}_{E_C} = \{o^r \mid o^r \in \mathcal{IS}_{E_C} \wedge o^r \notin \mathcal{O}_{E_C}\}.$$

The definition states that an object $o^r \in S_{E_C}$ is *distributed* if it is indirectly shared and it is not owned. That is, a distributed object is an object potentially indirectly accessed by multiple client threads; however, unlike an owned object, these accesses happen in a distributed manner, through multiple distinct objects.

6.1.3.2 Example

We illustrate these patterns with `jdbm`. Figure 6.1 shows excerpts from classes `BaseRecordManager`, `RecordFile` and `TransactionManager` in `jdbm`. Each `BaseRecordManager` object creates its `RecordFile` and `PhysicalRowIdManager` objects and assigns these objects to fields `_file` and `_physMgr` respectively. The `RecordFile` object, in turn, creates a `TransactionManager` object which it stores into its field `txnMgr`; the `TransactionManager` object points back to the `RecordFile` object — field `owner` is set to point to the creating `RecordFile` object. `BaseRecordManager`, designed as a thread-safe class, has most of its methods `synchronized`. We include methods `getTxnManager()` and `close()`.

Figure 6.2 shows a client of these classes which meets the constraints outlined in Section 6.1.1. Method `main` creates an instance of `BaseRecordManager`; it then calls method `getTxnManager` to obtain a reference of the transaction manager, and proceeds to call method `synchronizeLog` on the `TransactionManager` object, and method `close` on the `BaseRecordManager` object.

Figure 6.3 shows the object graph corresponding to Figures 6.1 and 6.2. Recall that an edge from object o_1 to object o_2 means that o_1 has access to o_2, or in other words o_1 holds a reference to o_2 and consequently, methods invoked on receiver o_1 may call methods on receiver o_2. Object $o_{BaseRecordManager}$ (created at creation site 4 in Figure 6.2) accesses objects $o_{RecordFile}$ (created at site 1 in Figure 6.1), $o_{PRowIdManager}$ (created at site 2 in Figure 6.1), and $o_{TransactionManager}$ (created at site 3 in Figure 6.1); the references to the first two are through fields

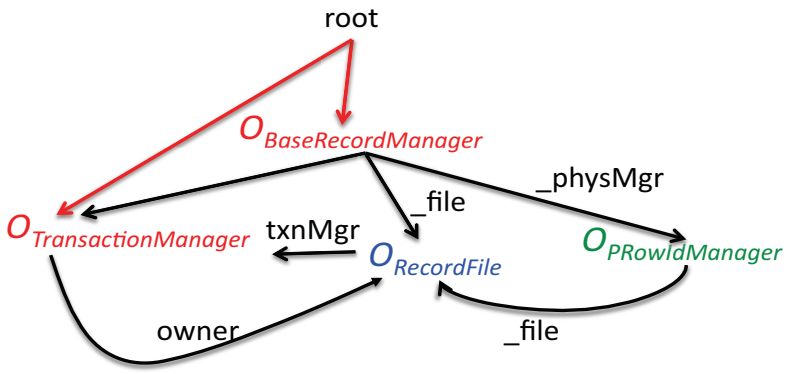

Figure 6.3: Object graph.

`_file` and `_physMgr`; the reference to the last is through `_file.txnMgr` in method `getTxnManager`. Method `main` of the client accesses objects $o_{BaseRecordManager}$ and $o_{TransactionManager}$; the first object one is created in `main`; the second object is returned to `main` by method `getTxnManager`.

In this example, object $o_{BaseRecordManager}$ is a central object — clearly, it is a shared object which is directly accessed by the client:

$$\texttt{root.main}() \rightarrow o_{BaseRecordManager}.\texttt{getTxnManager}()$$

Object $o_{TransactionManager}$ is central as well:

$$\texttt{root.main}() \rightarrow o_{TransactionManager}.\texttt{synchronizeLog}()$$

Object $o_{PRowIdManager}$ is an owned object because all accesses from `root.main`() to it go through $o_{BaseRecordManager}$. For example:

$$\texttt{root.main}() \rightarrow o_{BaseRecordManager}.\texttt{close}() \rightarrow o_{PRowIdManager}.\texttt{close}().$$

Finally, $o_{RecordFile}$ is a distributed object. It is an indirectly shared object; however, unlike the accesses to owned object $o_{PRowIdManager}$, the accesses to $o_{RecordFile}$ are distributed through two distinct objects:

$$\texttt{root.main}() \rightarrow o_{TransactionManager}.\texttt{synchronizeLog}() \rightarrow o_{RecordFile}.\texttt{synch}()$$

and

$$\texttt{root.main}() \rightarrow o_{BaseRecordManager}.\texttt{close}() \rightarrow o_{RecordFile}.\texttt{close}().$$

6.1.3.3 Discussion

Central objects are the objects potentially directly accessed by multiple threads (e.g., $o_{BaseRecordManager}$ is a central object). Therefore, their classes are typically synchronized with the intention to be "safe" when used by multithreaded clients.

In most applications, a central object creates and accesses a large number of objects during its lifetime. Some of these objects remain "hidden" behind the creating central object and all accesses to such "hidden" objects go through their creating central object (e.g., $o_{PRowIdManager}$ is "hidden" behind central object $o_{BaseRecordManager}$). Other objects become accessible to other central objects (e.g., $o_{RecordFile}$ is created by central object $o_{BaseRecordManager}$ but eventually it becomes accessible to central object $o_{TransactionManager}$ as well). These indirectly accessed objects, both the ones that are hidden and the ones that are not, can exhibit object and data races.

Owned objects are hidden behind an "owner" object, typically one of the central objects. Distributed objects are the shared objects not hidden behind an "owner" object — they can be accessed in a distributed manner, through several central objects.

We conjecture that there are two important benefits from these patterns. First, the patterns facilitate general-purpose program understanding. They provide a simple and intuitive classification of the shared objects in concurrent programs with complex shared structures, which may lead to better program understanding, better testing, better detection and *more effective correction* of concurrency errors. Second, the patterns facilitate concurrency error detection. They classify the shared objects into two categories: "easy to reason about" and "difficult to reason about" objects; an error detection techniques can examine "easy" objects quickly, and focus on "difficult" objects thereafter.

Consider the problem of reasoning about *object races* (informally, an object race on an object o occurs when two threads can access o "simultaneously").

Central objects are "easy to reason about". A central object o_c is accessed

directly by multiple threads calling methods on o_c; any pair of methods called on o_c, say $o_c.m_1()$ and $o_c.m_2()$, where one of m_1 or m_2 is unsynchronized, create a potential object race on o_c.

Owned objects are "easy to reason about" as well. An owned object o_o has an "owner" object o_c. There are two important benefits from ownership. First, often an owned object is protected by synchronization on its owner; that is, the owned object is "safe" from object races because of its owner. In our running example $o_{PRowIdManager}$ is protected by synchronization on its owner $o_{BaseRecordManager}$; it is guaranteed that no object races occur on $o_{PRowIdManager}$. Second, even if the owned object is not protected by synchronization on its owner o_c, reasoning about object races is simplified by ownership: the detection technique must first identify an object race on o_c, and if such a race is identified, proceed to search for an object race on o_o *exclusively* within the ownership boundary of o_c. Such structured detection can reduce search space and time.

On the other hand, distributed objects are "difficult to reason about". A distributed object o_d may be accessed through multiple central objects — generally, a search for object races on o_d will need to traverse the entire set of objects which could be quite large. In our example, $o_{RecordFile}$ is a distributed object — it can be accessed along several paths: e.g., `root` $\rightarrow o_{BaseRecordManager} \rightarrow o_{PRowIdManager} \rightarrow o_{RecordFile}$ and `root` $\rightarrow o_{TransactionManager} \rightarrow o_{RecordFile}$.

We conjecture that these patterns have important software engineering benefits: first, they facilitate general-purpose program understanding, and second, they facilitate concurrency error detection. We claim that the role of ownership in concurrent programs should be enhanced — well-written concurrent programs should strive for a large number of owned (i.e., "easy to reason about") objects and a minimal number of distributed (i.e., "difficult to reason about") objects. Whenever possible, programmers should refactor code to hide distributed objects behind an owner object.

6.1.4 Preliminary Analysis

The goal of this work is to design a static analysis that infers central, owned and distributed objects across all execution of clients $C \in AllClients(Int)$ of *Cls*. We proceed by performing five preliminary analyses — (1) fragment analysis (described in Section 4.1), (2) points-to analysis (described in Section 4.2), (3) object graph construction (described in Section 4.3.1), (4) ownership analysis (described in Section 4.3.2), and (5) method sequence analysis (will be discussed in Section 6.1.4.1).

The input is a set of classes *Cls* and the analysis needs to reason across all possible executions of arbitrary client code built on top of *Cls*. To address this problem we use the *fragment analysis* to enable analysis on partial programs. The placeholder `main` for the classes from Figure 6.1 is shown below:

```
public class PlaceholderClient {
   public static void main() {
      BaseRecordManager ph_brm = new BaseRecordManager();
      RecordFile ph_rf = new RecordFile();
      TransactionManager ph_tm = new TransactionManager();
      ph_tm = ph_brm.getTxnManager();
      ph_brm.close();
      ph_tm.synchronizeLog();
   }
}
```

6.1.4.1 Method Sequence Analysis

Method sequence analysis approximates the run-time method sequence graphs over executions of clients $C \in AllClients(Int)$.

Recall from Section 6.1.2 the definition of the run-time method sequence graph for an execution of a client $C \in AllClients(Int)$. The method sequence analysis constructs the static method sequence graph, Og^+. The nodes in Og^+ are tuples $o_i.m_i()$ (the tuples are formed with analysis objects), and the edges represent the transfer of control between distinct objects. Og^+ is a safe approximation of the run-time method sequence graphs over all executions of all clients of *Cls*: if there

procedure *computeMethodSequences*
uses Og
output Og^+
[1] foreach statement $l.m_2()$, $l \neq$ `this`, in method m_1
[2] foreach $o_1 \rightarrow o_2 \in Og$, s.t., $(o_1, \texttt{this}_m) \in Pt \wedge (o_2, l) \in Pt$
[3] $m_2 = dispatch(l.m_2(), o_2)$
[4] add $o_1.m_1() \rightarrow o_2.m_2$ to Og^+

[5] while Og^+ changes
[6] foreach $\texttt{this}.m_1()$ in method m_1'
[7] foreach $o_1.m_1() \rightarrow o_2.m_2() \in Og^+$, s.t. $(o_1, \texttt{this}_{m_1'}) \in Pt$
$\wedge \;\; m_1 = dispatch(\texttt{this}.m_1(), o_1)$
[8] add $o_1.m_1'() \rightarrow o_2.m_2$ to Og^+

Figure 6.4: Method sequence analysis.

is an execution that exhibits method sequence $o_i^r.m_i() \rightarrow o_j^r.m_j()$, then there is a representative method sequence $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$.

The method sequence analysis uses the constructed object graph Og and points-to graph Pt, and outputs the method sequence graph Og^+.

Figure 6.4 presents the method sequence analysis. Lines 1-4 add edges due to calls not through `this`; this part of the analysis corresponds to the first bullet in Section 6.1.2. Lines 5-8 proceed to compute the closure of Og^+ by considering calls through `this`; this part corresponds to the second bullet in Section 6.1.2.

Line 1 identifies call statements $l.m_2()$, $l \neq$ `this`; these statements trigger method sequences between distinct objects. Line 2 identifies edges $o_1 \rightarrow o_2 \in Og$ affected by $l.m_2()$—these are the edges where o_1 is a receiver of the enclosing method m_1, and o_2 is a receiver at the call $l.m_2()$. Line 3 identifies method m_2 — the run-time target dispatched at call site $l.m_2()$ with receiver o_2. Line 4 adds $o_1.m_1() \rightarrow o_2.m_2()$ to the method sequence graph Og^+.

Line 6 examines call statements $\texttt{this}.m_1()$; these statements may result in edges that must be propagated to the enclosing method m_1'. Line 7 checks if there is an edge $o_1.m_1() \rightarrow o_2.m_2()$ in Og such that o_1 is in the points-to set of `this` and m_1 is the run-time target dispatched at call site $\texttt{this}.m_1()$. Line 8 adds $o_1.m_1'() \rightarrow o_2.m_2()$ to the method sequence graph Og^+ (i.e., it propagates the call to $o_2.m_2()$ which result

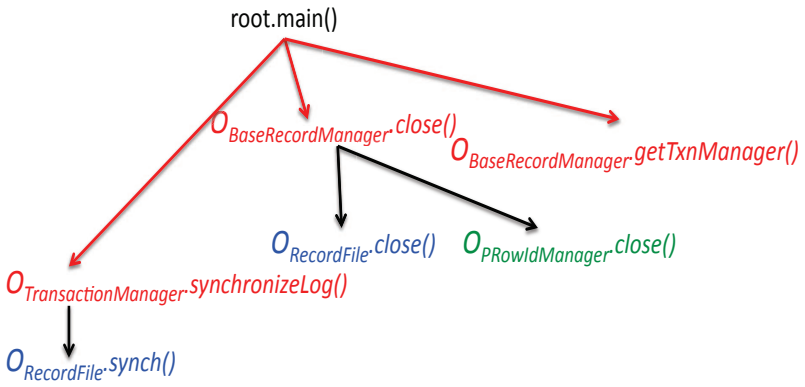

Figure 6.5: Method sequence graph Og^+ for code in Figures 6.1 and 6.2.

from the call `this`.$m_1()$, to the enclosing method m_1').

Consider our running example from Figures 6.1 and 6.2. Consider call site `_file.close()` and site `_physMgr.close()` in method `close` in `BaseRecordManager` class. This call site results in the following edge:

$$o_{BaseRecordManager}.close() \rightarrow o_{PRowIdManager}.close()$$

The method sequence graph for this example is shown in Figure 6.5; the corresponding object graph is shown in Figure 6.3. Constructors are not shown for brevity.

6.1.5 Inference of Patterns of Object Sharing

The inference of patterns of object sharing is shown in Figure 6.6. It takes as input the method sequence graph Og^+ discussed in Section 6.1.4.1, boundary information *Boundary* described in Section 4.3.2, and outputs sets *CENTRAL*, *OWNED* and *DISTRIBUTED*. The algorithm is a breadth-first search on Og^+ starting at tuple `root.main`(), where `root` is the thread object created in `main` of the artificial client. The analysis examines all shared objects; these are the objects o such that a tuple with receiver o, say $o.m()$, is reachable in Og^+ from `root.main`(): we have `root.main`() $\rightarrow^*$ $o.m() \in Og^+$ (this approximates run-time reachability on the method sequence graph `root.main`() $\rightarrow^*$ $o^r.m()$). Lines 5-10 of the analysis algorithm contain the code for visiting a tuple $o_j.m_j()$, line 11 marks the tuple as visited, and line 12 adds the tuple at the end of worklist *WL*. Lines 5-6 identify

procedure *inferPatterns*
global Og^+, *Boundary*
output sets *CENTRAL*, *OWNED* and *DISTRIBUTED*
[1] $WL = \{$`root.main()`$\}$
[2] while WL is not empty
[3] take $o_i.m_i()$ from WL
[4] foreach $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$ and $o_j.m_j()$ not visited
[5] if $o_i ==$ `root`
[6] add o_j to *CENTRAL*
[7] else if $\exists\ o_k$ s.t. $o_k \neq$ `root` $\wedge\ o_i \rightarrow o_j \in Boundary(o_k)$
[8] add o_j to *OWNED*
[9] else if $o_j \notin$ *CENTRAL*
[10] add o_j to *DISTRIBUTED*
[11] mark $o_j.m_j()$ as visited
[12] add $o_j.m_j()$ at the end of WL

Figure 6.6: Inference of objects sharing patterns.

the central objects. Lines 7-8 identify the owned objects. Lines 9-10 identify the distributed objects.

Consider the example from Figures 6.1 and 6.2, and the method sequence graph in Figure 6.5. The algorithm examines edge

$$\texttt{root.main}() \rightarrow o_{TransactionManager}.synchronizeLog().$$

Tuple $o_{TransactionManager}$`.synchronizeLog()` is visited at lines 5-10 and $o_{TransactionManager}$ is added to *CENTRAL*. Subsequently, $o_{TransactionManager}$`.synchronizeLog()` is marked as visited and it is added at the end of the worklist. Next, the algorithm examines edges

$$\texttt{root.main}() \rightarrow o_{BaseRecordManager}.close()$$

and

$$\texttt{root.main}() \rightarrow o_{BaseRecordManager}.getTxnManager().$$

$o_{BaseRecordManager}$ is added to *CENTRAL*, and tuples $o_{BaseRecordManager}$`.getTxnManager()` and $o_{BaseRecordManager}$`.close()` are marked as visited and added at the end of the worklist.

Then $o_{TransactionManager}$.`synchronizeLog()` is taken from the worklist at line 3. As a result, the algorithm examines edge

$$o_{TransactionManager}.synchronizeLog() \rightarrow o_{RecordFile}.synch().$$

At lines 7-8 the algorithm examines object graph edge

$$o_{TransactionManager} \rightarrow o_{RecordFile}$$

and determines that there is no o_k, besides `root`, such that the edge is in the dominance boundary of o_k. In other words, there is no "owner" o_k that could protect $o_{RecordFile}$. The algorithm determines that $o_{RecordFile}$ is distributed and adds it to *DISTRIBUTED*. Tuple $o_{RecordFile}$.`synch`() is marked as visited and added at the end of the worklist.

Next, $o_{BaseRecordManager}$.`close()` is taken from the worklist at line 3, As a result, the algorithm examines edge

$$o_{BaseRecordManager}\, close() \rightarrow o_{PRowIdManager}.close().$$

Again, at lines 7-8 the algorithm examines object graph edge

$$o_{BaseRecordManager} \rightarrow o_{PRowIdManager}$$

and in this case the edge is in the dominance boundary of $o_{BaseRecordManager}$. Clearly, the "owner" object $o_{BaseRecordManager}$ can protect $o_{PRowIdManager}$. The algorithm determines that $o_{PRowIdManager}$ is owned and adds it to *OWNED*. Tuple $o_{PRowIdManager}$.`close`() is marked as visited and added at the end of the worklist.

The analysis algorithm proceeds until the worklist *WL* is empty. The final result is:

$CENTRAL = \{o_{BaseRecordManager}, o_{TransactionManager}\}$
$OWNED = \{o_{PRowIdManager}\}$
$DISTRIBUTED = \{o_{RecordFile}\}$.

Program	Description	#Reachable Methods
`jdbm`	transactional persistence engine	4904
`jdbf`	object-relational mapping system	6383
`pool`	Apache Commons pool	3982
`jtds`	JDBC driver	5980

Table 6.1: Information about the benchmarks.

Program	*OWNED*	*CENTRAL*	*DISTRIBTUED*	Analysis time
`jdbm`	22	23	32	142 seconds
`jdbf`	38	41	19	522 seconds
`pool`	8	35	13	84 seconds
`jtds`	100	74	272	1527 seconds
`Average`	25%	39%	36%	

Table 6.2: Results on object sharing patterns.

6.1.6 Empirical Study

We applied our framework of ownership analysis for inference of patterns in Sections 6.1.5. The inference of patterns is implemented in Java using the infrastructure of our framework. We performed the analysis with the Sun JDK 1.4.1 libraries. The implementation, which includes Soot and Spark, was run with a max heap size of 1400MB.

We used four publicly available open concurrent Java programs: `jdbm` 1.0, `jdbf`, `jtds` 1.2 and `pool` 1.2. Except for `jdbf`, which is under development, the rest are mature applications. Information about these programs is given in Table 6.1. Column 2 describes the benchmark, and Column 3 gives the number of methods determined as reachable by the points-to analysis in Spark. These programs were used in work on static race detection [22].

Table 6.2 shows the results of the analysis for inference of patterns, and the analysis for inference of object races. Columns 2, 3 and 4 list the sizes of sets *OWNED*, *CENTRAL* and *DISTRIBUTED* for each of our benchmarks. The majority of shared objects, on average 39%, are central; about a quarter, on average 25%, are owned, and about a third, on average 36%, are distributed.[15]

[15]Note that in terms of absolute numbers, our analysis is very conservative — that is, it creates a large number of central objects, and in turn these central objects lead to a large number of distributed and owned objects (recall that each public class in *Cls* is instantiated in the artificial client). In typical clients, there would be a smaller number of central objects, and in turn there

Our results show that a significant percentage of shared objects, about two thirds, or 64%, are owned or central. This can have significant impact on a conservative search for object and data races (i.e., search that takes each possible client of *Cls* into account) —- we have that the vast majority of objects are easy to reason about (and prove as "thread-safe" or "thread-unsafe").

Our results show that a significant percentage of objects, 27%, are owned.These results show that ownership does play a role in sharing in concurrent programs. Again we claim that this role should be enhanced — well-written multi-threaded programs should strive for a large number of owned objects and a minimal number of distributed objects.

The running time of the analysis is shown in the last column of Table 6.2; this includes the running times for (1) the object graph construction analysis (described in Section 4.3.1), (2) the ownership inference analysis (in Section 4.3.2), (3) the methods call analysis (in Section 6.1.4.1), and (4) the inference of patterns (in Section 6.1.5). In all cases, this running time is heavily dominated by ownership inference; the inference of patterns take only seconds.

6.2 Applications of Information Flow Analysis

Static information flow inference analysis has a wide variety of applications. It can be utilized (1) for *security violation detection* — that is, the detection of security attacks and security vulnerabilities, (2) for *type inference* for security type systems, and (3) for the purposes of *general program understanding*,

For security violation detection, information flow inference can help detect security attacks such as script, web cache, web page, and SQL injection attacks; it can help detect security vulnerabilities that release confidential data to untrustworthy parties (e.g., echoing a password on the screen). Security type systems such as JFlow [23], require that variables and statements are annotated with security types (labels which denote security levels). Type inference based on information flow inference analysis can lessen the burden of manually writing type annotations, and can greatly facilitate the application of security type systems in software practice. In

will be a smaller number of owned and distributed objects.

(1)Benchmark	(2)Version	(3)#Sources	(4)#Sensitive data	(5)#Rechable Methods	(6)#Security violations	
					Explicit	Incl. Implicit
`jboard`	0.30	1	16	4220	0	0
`blueblog`	1.0	11	39	4836	1	8
`webgoat`	0.9	10	81	5698	9	16
`personalblog`	1.2.6	31	32	9570	0	0
`pebble`	1.6-beta1	124	78	7622	1	1
`roller`	0.9.9	40	94	13623	1	1

Table 6.3: Security violations in Web application security benchmarks.

addition, security type systems are notorious for uninformative error messages when type checking fails. Information flow inference analysis can be used to help provide more helpful information that shows *why* and *where* security type checking fails. It can also help guide programmers to places where *declassification* and *endorsement* statements are needed — these statements temporarily relax the program's security constraints, and are known to be difficult to get right. For general program understanding, information flow analysis can illustrate how information is transmitted through assignments, method calls and control flow statements; information flow can be used to reason about multithreaded programs.

To illustrate the usage of our static information flow analysis described in Section 4.5, we applied our framework of information flow analysis on several applications. The implementation, which includes Soot and Spark, was run with a max heap size of 1000MB.

6.2.1 Security Violation Detection

For security violations, our analysis is performed on a set of web applications, SecuriBench [24], established as security benchmarks[16]. `jboard`, `blueblog`, `personalblog`, `pebble`, and `roller` are web-based bulletin board and blogging applications. `webgoat` is a J2EE application designed as a test case and a teaching tool for Web application security.

The benchmarks are described in the first three columns of Table 6.3. The version of each benchmark is shown in Column (2). The possible *sources* for se-

[16]The set consists of 8 applications. However, application `blojsom` from the suite is no longer available; we included all other applications except for one, `snipsnap`, which ran out of memory through our Soot-based infrastructure.

curity attacks are untrusted inputs such as fields including hidden fields of HTML forms, submitted URLs, received HTTP requests, obtained cookies, etc (identified according to [24]). The number of these sources is shown in Column (3). Sensitive data for possible security attacks are variables that must be trusted, such as parameters passed to particular methods, such as SQL queries, written values of server-side output streams, commands executed by the system, file paths or values sent as http response. The number of sensitive variables is shown in Column (4) (identified according to [24]). Column (5) shows the number of methods, including library methods, determined to be reachable by Spark.

Therefore, the analysis examines information flows from identified sources to identified sensitive data. If no such information flow exists according to the analysis, then it is guaranteed that there are no security attacks of the specified kind. If the analysis identifies information flow, then this constitutes a potential security attack. The results of our analysis are shown in Column (6). Each reported security violation is a pair (p, s), where p is a *source*, and s is an identified sensitive variable, and there is information flow from p to s; in other words, s is not safe.

The analysis of `webgoat` and several other applications revealed a vulnerability in the core J2EE libraries, the `doTrace` method specified in the HTTP protocol. The `doTrace` method echoes the contents of an HTTP request back to the client for debugging purposes. However, there exists a security violation, from source *HttpServletRequest.getHeader*(*headerName*) to sensitive data *responseString* which is printed by *ServletOutputStream*; thus, the contents of user-provided headers are sent back verbatim, enabling cross-site scripting attacks.

We examine the impact of implicit flow on this application. First, we run the version of the analysis which creates $\mathcal{FG}_0$ with explicit flow edges only. The number of detected security violations is shown in the first sub-column of Column (6) in Table 6.3. Next, we run the version analysis which creates $\mathcal{FG}_0$ with both explicit and implicit flow edges. The number of detected violations is shown in the second sub-column of Column(6). Implicit flow has no impact on four benchmarks (i.e., it does not detect additional security violations compared to explicit flow). However, on *blueblog* and *webgoat*, it detects 7 additional security violations per benchmark.

(1)Benchmark	(2)#Untrusted variables (Explicit)	(3)#Untrusted variables (Including implicit flow)	(4)#Time
`jboard`	1	1	15s
`blueblog`	217	5733	81s
`webgoat`	182	34410	177s
`personalblog`	892	11438	213s
`pebble`	531	611	78s
`roller`	3016	3209	508s

Table 6.4: Type inference for untrusted variables.

Implicit flow captures more security violations; our examination confirms that the reported violations are not false positives.

We compared the results of our explicit flow analysis with the results reported by the explicit flow analysis in [24]. Our results are the same as reported in [24], except for 2 cases. In application `webgoat`, we discovered 3 more security violations than were reported in [24]; our manual examination confirmed these security violations. Second, our analysis did not discover the 2 violations in application `personalblog`, due to utilized library `hibernate 2.1.4`. That 2 violations are missed by our analysis may be due to the smaller number of sensitive data defined by our analysis[17].

The time for security violation detection is shown in the last column of Table 6.4. Overall, the precision experiments and the analysis time confirm that our inexpensive analysis achieves very good results.

6.2.2 Type Inference

Security type systems, such as JFlow [23], require that variables and statements are annotated with security types (labels which denote security levels). Type inference based on information flow inference analysis can lessen the burden of manually writing type annotations, and can greatly facilitate the application of security type systems in software practice.

We perform type inference on the same set of security benchmarks described

[17]Though we use the same standard for selection of sensitive data as [24], our analysis identified 7 sensitive variables while their analysis identified 31. This difference might be due to different versions of the imported libraries, and different amounts of analyzed code which are decided by different underlying points-to analysis.

(1)Benchmark	(2)#Reachable Methods	(3)#Thread -shared Variables	(4)#Affected Thread-local Variables	
			Explicit	Incl. Implicit
`hedc`	4403	36	420 (31.8%)	912 (69.1%)
`sor`	3578	4	154 (69.4%)	212 (95.5%)
`tsp`	3415	15	254 (74.5%)	323 (94.7%)
`montecarlo`	3526	17	204 (71.3%)	267 (93.3%)
`raytracer`	3481	4	201 (36.7%)	380 (69.3%)

Table 6.5: Effects of thread-shared variables on thread-local variables.

in Section 6.2.1. The security type being inferred is the *untrusted* type. The initial set of *untrusted* type variables are variables identified as *sources* described in Section 6.2.1.

For each program, the analysis propagates the *untrusted* type labels from the initial set. This essentially is a taint analysis — it identifies flow from the untrusted *sources* specified in Table 6.3, to all other variables in the program. The number of inferred untrusted type variables is shown in Table 6.4.

We examine the impact of implicit flow on this application as well. First, we run the version of the analysis which creates $\mathcal{FG}_0$ with explicit flow edges only. The number of inferred *untrusted* type variables is shown in Column (2) of Table 6.4. Next, we run the version analysis which creates $\mathcal{FG}_0$ with both explicit and implicit flow edges. The number of inferred *untrusted* type variables is shown in Column (3). As seen from Table 6.4, implicit flow has significant impact on type inference, as thousands of additional *untrusted* variables are detected after considering implicit flows.

Column (4) in Table 6.4 shows the combined running time for security violation detection and type inference. The running time illustrates that the analysis is practical — it runs within 1 to 4 minutes for most benchmarks, except for `roller` on which it runs in around 8 minutes.

6.2.3 Effect of Thread-shared Variables

We apply our analysis to study the effect of *thread-shared* variables on *thread-local* variables. The analysis is performed on a suite of multi-threaded benchmarks which includes programs from the Java Grande suite (`montecarlo` and `raytracer`),

(1)Benchmark	(2)#Thread -shared	(3)#Affected Fields		(4)#Total Time
		Explicit	Incl. Implicit	
`hedc`	36	30 (51.7%)	58 (100%)	24s
`sor`	4	4 (40%)	4 (40%)	16s
`tsp`	15	9 (75%)	11 (91.7%)	16s
`montecarlo`	17	36 (87.8%)	39 (95.1%)	14s
`raytracer`	4	4 (8.33%)	20 (41.7%)	14s

Table 6.6: Effects of thread-shared variables on instance fields.

and programs from ETH (a Traveling Salesman Problem solver `tsp`, a successive over-relaxation benchmark `sor`, and a web crawler `hedc`). Column (2) in Table 6.5 shows the number of reachable methods for each benchmark, including library methods, determined by Spark.

The set of *thread-shared* variables includes all static fields that are accessed in methods reachable from any `run` method. The number of the thread-shared variables is shown in Column (3) of Table 6.5. The set of *thread-local* variables includes all local variables declared in methods reachable from any `run` method, and all instance fields that are written within a thread.

Our analysis examines the effect of thread-shared variables by tracking information flows from these thread-shared variables. If there exists information flow from a thread-shared variable to a thread-local variable, we say that the thread-local variable is affected by the thread-shared variable; this states that the value of the thread-local variable is dependent on that thread-shared variable. The number and percentage of affected thread-local variables are shown in Column (4) of Table 6.5. The number and percentage of affected written instance fields are shown in Column (3) of Table 6.6. These affected variables and fields need to be examined carefully, because they depend on the values of thread-shared variables.

As with the other applications, we examine the impact of implicit flow. First, we run the version of analysis which creates $\mathcal{FG}_0$ with explicit flow edges only. The first sub-column of Column (4) in Table 6.5 shows that on average a third of the thread-local variables are affected by thread-shared variables. The first sub-column of Column (3) in Table 6.6 shows that on average half of the written instance fields are affected by thread-shared variables. Second, we run the version of the analysis

which considers both explicit and implicit flow. The results show that implicit flow significantly impacts this application. The second sub-columns of Column (4) of Table 6.5 and Column (3) of Table 6.6 show that implicit flow captures at least 20% additional affected thread-local variables and 10 to 40% additional affected written fields.

Column (4) in Table 6.6 shows the running time of the analysis on each program. The analysis runs within 30 seconds on all the benchmarks.

CHAPTER 7
Related Work

The ownership and immutability inference analyses improve substantially upon similar previous work on composition inference [9] and side-effect analysis [15]. The main new analysis idea is to employ an inexpensive context-insensitive points-to analysis and improve precision by limited context sensitivity in the clients. This was crucial for precision and scalability; in fact, the old analysis was not only potentially imprecise, but it did not scale beyond the smallest benchmarks in our suite. Further, the analyses could be employed towards a new practical purpose—improving the capabilities of UML tools, which will enhance object access control and thus software security and software quality in practice.

There are many proposals for language-based reasoning about ownership, immutability and information flow—there are proposals for ownership type systems (e.g., [25, 2, 26]), immutability type systems (e.g., [27]) and type systems for secure information flow (e.g., [23, 28]). Similarly to our work, this work emphasizes the importance of the concepts of ownership, immutability and information flow in software development. Unlike our work, it focuses on type-theoretic approaches which in general require extensions to the language, compiler and run-time environment, as well as type annotations provided by the programmer. Therefore it may be difficult to adopt these approaches in practice.

Somewhat surprisingly, automatic inference of ownership, immutability and information flow has received significantly less attention. Recent work on inference of ownership-like properties includes [29, 30, 31, 32] as well as the work by Gueheneuc and Albin-Amiot [33] on dynamic analysis for inference of aggregations and compositions for reverse-engineered UML class diagrams. Work on inference of immutable parameters and methods includes [34, 8, 35], and work on inference of information flow includes [10].

7.1 Type systems

Our work is related to work on ownership type systems [25, 2, 26, 6, 36, 37] and work on immutability type systems [38, 39, 27]. Similarly to our work, these articles emphasize the importance of the concepts of ownership and immutability in software development. Unlike our work they focus on type-theoretic approaches and require type annotations provided by the programmer; generally, these approaches require extensions of the language, compiler and run-time environment and therefore will be difficult to adopt in practice. In contrast, our approach uses automatic inference and works directly on Java code; it is based on the universally-known UML and therefore may help advance object access control through ownership and immutability in practice.

There are approaches rely on type systems for secure information flow [40, 41, 23, 42, 43]. Generally, these approaches require changes to the language, compiler and run-time system, as well as sometimes complex type annotations provided by the programmer; therefore, it may be difficult to adopt these approaches in practice. In contrast, our analysis works directly on Java codes and does not require annotations; it can be directly incorporated in program understanding and verification tools.

7.2 Ownership inference

Grothoff et al. [44] present an analysis for Java that infers whether a class is confined within its package. Clarke et al. [45] present a confinement checking tool, related to [44], that warns against certain kinds of violating program statements. These analyses work on the class level while our analyses work on the object level. They are more restrictive than ours (e.g., they do not handle generic containers well), and do not address the kind of ownership needed for UML-based object access control.

Heine and Lam [46] present an ownership inference algorithm for the purposes of memory leak detection. Their notion of ownership is based on memory management, in which the ownership can be transferred. This is substantially different than the notion of owners-as-dominators used in our work.

Aldrich et al. [26] present a type inference analysis in accordance with a type

system that they develop. Again, our analysis solves a different problem—ownership inference in accordance with the owners-as-dominators model which is different than the type system in [26] (e.g., the `owned` type in [26] captures exclusive ownership only, although access can be allowed through user-specified alias parameters). The inference analysis in [26] is conceptually different than ours; it infers type annotations at a fine level of granularity (i.e., for each variable and expression) and that appears to hinder scalability. Our analysis, which is based on Soot, and the efficient inclusion-based Andersen-style points-to analysis in Spark, appears to scale substantially better, both in terms of time and memory.

Agarwal and Stoller [47] infer ownership types for race-free Java using dynamic analysis; thus, the inferred types may be unsound. In contrast, our inference analysis is a sound static analysis.

Recent work by Rayside et al. [48] emphasizes the relevance of ownership inference and visualization. The paper, however, appears to be preliminary because it does not present empirical results. Our work uses a related ownership model, but a conceptually different inference analysis. It presents a detailed empirical investigation that indicates that the analyses are practical and adequately precise.

7.3 Immutability inference

Porat et al. [49] describe an analysis that detects immutable fields. Their analysis is context-insensitive, libraries are not analyzed and the paper discusses only static fields. Our immutability analysis incorporates limited context sensitivity, analyzes large libraries and focuses on instance fields.

Ryder et al. [14] present a framework for side-effect analysis for C that is parameterized by points-to analysis. Our inference analysis uses the same general idea for propagation of side-effects. However, we consider underlying context-insensitive points-to analysis combined with limited context sensitivity during propagation; this combination helps achieve scalable analysis.

Rountev [8] and Salcianu and Rinard [34] present analyses that identify side-effect-free methods in Java programs. In both cases the analyses are applied on relatively small programs (hundreds of reachable methods). Our analysis identifies

immutable fields and is applicable on substantially larger programs (close to ten thousand reachable methods).

7.4 Static information flow analysis

Genaim and Spoto [10] present an information flow analysis for Java bytecode. Their analysis works on complete programs only and does not separate flow through fields of different objects which may lead to significant imprecision; in contrast, our analysis works both on complete and incomplete programs and separates flow through different object fields. Furthermore, our analysis is different: it is cubic and based on CFL-reachability, which we conjecture achieves scalability and precision for this problem. Finally, we present results on absolute precision which indicate that our analysis may achieve better precision.

Livshits et al. [24, 50] propose analysis for finding vulnerabilities caused by unchecked inputs. This analysis requires users to provide vulnerabilities patterns, while our analysis is automatic. Furthermore, it only tracks flow of objects, while our analysis considers flow for both object and simple types. Again, our analysis is different: the analysis in [24, 50] is exponential (due to the underlying points-to analysis), while ours is cubic.

Tripp et al. [51] propose a static taint analysis for Java that detects security vulnerabilities by reasoning about information flows. They first construct a context-sensitive points-to analysis, and then use a hybrid algorithm for slice construction to track tainted data. They focus on explicit flows only, not considering information flows through control dependencies, while our analysis considers both explicit and implicit flows.

Horwitz et al. [52] define the standard system dependence graph to perform the context-sensitive slicing. Their graph consists of edges among expressions, to capture dependencies due to flows, while our approach captures flows with edges among variables. Their approach import call context by computing modified variables triggered by calling a method, and referenced variables triggered by calling that method, while we use extended CFL-reachability for context-sensitivity. They suggest two approaches to deal with aliasing, one is to create copies of procedure

for each alias configuration, which is exponential; the other is to generalize the flow dependence as under possible aliasing patterns, which is polynomial but the precision and scalability remain unknown; while our approach uses the relatively precise and scalable Andersen's points-to analysis.

Scholz et al. [53] introduce a static program analysis for computing user-input dependencies in programs based on Static Single Assignment form. Similarly to our approach, they construct a directed graph representing either data or control dependencies, and compute graph reachability to decide user-input dependencies. Their analysis does not deal with objects, being aimed for C programming language family, and requires a configuration file for libraries, while our analysis deals with objects and fields in Java and does not require annotations in libraries.

Hammer et al. [54] present a precise approach to information flow control based on a combination of dependence graphs and constraint solving. Their dependence graphs captures dependencies between statements or expressions, while our approach captures dependencies between variables. Their preliminary study includes just two benchmarks and thus does not show the scalability of their approach, while our empirical investigation on a Java web application suite and a multi-threaded application suite demonstrates scalability.

Related type inference techniques have been proposed [55, 56, 57]. One disadvantage of these techniques is that they lack support for libraries: they either require users to provide type annotations for libraries, or restrict the usage of libraries. In contrast, our analysis handles libraries seemlessly: it analyzes reachable library code and tracks flow through this code.

7.5 Dynamic information flow tainting

Dynamic tainting labels data and propagates the labels during execution through suitable instrumentation. There are tainting-based tools that prevent integrity-compromising attacks on network services [58, 59, 60], tools that detect SQL-injection attacks [61, 62, 63], and tools that enforce data confidentiality [64, 65, 66, 67]. Recently, Clause et al. have proposed a general framework for dynamic tainting [68]. Dynamic tainting is a conceptually different approach to secure in-

formation flow: it tracks flow through instrumentation during execution, while our analysis tracks flow statically, before program execution.

7.6 CFL-reachability

CFL-reachability is a well-known technique for context-sensitive program flow analysis [16]; it has been used in a variety of flow analyses that require context sensitivity (e.g., points-to analysis for Java [19], and analysis for race detection for C [69]). Our analysis is a CFL-reachability computation as well; one can see that the *concat* operations are essentially grammar productions. We conjecture that CFL-reachability presents the suitable degree of scalability and precision for the problem of static information flow analysis.

Our work builds on the ideas in Rehof and Fahndrich [17]. Unlike [17], it deals with non-structural (i.e., inclusion-based) flow and it needs to consider flow through object fields, which is a known problem: analysis that tracks flow through fields *and* flow through method contexts precisely is undecidable [70], and one needs an approximation at least in one of these dimensions. Our analysis approximates flow through fields and seamlessly weaves the approximation into the reachability computation by using $*$-annotations; one can vary the degree of approximation by varying the precision of the underlying points-to analysis, while the client analysis remains the same (and cubic).

7.7 Our approach

The main novelty of our approach is that it focuses on the inferences of important security-related properties. We present a new extensible static analysis framework which allows automatic inference of different kinds of properties. Finally, we focus on analysis precision, and carefully study precision and scalability; our results indicate that our analysis framework may allow for better precision and scalability than previous static analyses.

CHAPTER 8
Conclusions and Future Work

Current languages such as Java do not provide effective mechanisms for reasoning about program security properties. Then software implementations may have violations against these security properties. These unintended violations may cause serious quality and security concerns. Thus it is important to study mechanisms for reasoning about program security properties. Most current work is either dynamic approaches which typically incur significant run-time overhead, or language-based approaches which typically require changes to the language and runtime as well as non-trivial type annotations. To achieve the goal of practical and relatively precise reasoning about security properties, we have proposed the use of a general framework for inference of implemented security properties.

8.1 Framework for Inference of Security-related Properties

We have proposed a general framework of practical static analyses reasoning about security-related program properties. The key idea is to illustrate the security-related properties at the design level as constraints on the class diagrams, define runtime models to capture the runtime behaviors of these properties, and use static analyses to automatically retrieve the implemented properties from the code, according to the runtime models.

The framework could be augmented with different runtime models and corresponding analyses to infer different security-related properties. Specifically, we have reasoned about *ownership*, *immutability* and *information flow* for the purpose of revealing information about object access, protection of data confidentiality and data integrity in the program, and helping uncover serious vulnerabilities.

The definition of implementation-level ownership was based on owners-as-dominators, that is, all access paths to an owned object should pass through its owner. The definition of immutability required that an enclosing object have read-only access to an enclosed immutable object, that is, the methods invoked on the

enclosing object cannot change (directly, or through called methods) the heap structure rooted at the enclosed immutable object. The definition of information flow was based on data dependency and flow dependence, that is, there is information flow from variable x into variable y, if changes in the input values of x are observable from the output values of y.

Ownership, immutability and information flow analyses have been designed according to the runtime models. These analyses worked directly on Java code and did not require annotations by the programmer. The analyses were practical and had polynomial time complexity. The complexity of ownership analysis was $O(N^4)$, and the complexity of immutability and information flow analyses were $O(N^3)$. It is important to note that the analyses worked both on complete programs and on incomplete programs (i.e., software components). This is an important feature because reasoning about security should be performed on software components such as libraries; any realistic program understanding and verification tool should be able to work on software components.

8.1.1 Practical and Precise Analyses Results

The static analysis framework has been evaluated on both software component benchmarks and complete program benchmarks. The empirical study on software components consisted of a set of Java components. The empirical study on complete programs consisted of several small-to-large Java applications. In our experiments of complete programs, on average 26% of the reverse-engineered associations were determined to be `owned`, 29% were determined to be `read-only`, and 43% of the examined data were determined to be `deep leaked`, 23% were determined to be `deep tampered`.

We have present a precision evaluation by manually examining the reported results. The evaluation indicated that the analyses achieve adequate precision—the ownership inference almost never missed an `owned` property, the immutability inference rarely missed a `read-only` property, and almost all identified confidentiality and integrity violations could actually happen.

On software components which had about 10 functionality classes and 3000

analyzed reachable methods, all the analyses ran within 1 minute on each benchmark, mostly within 10 to 20 seconds. On relatively large complete programs which have about 4000 to 9000 analyzed methods, the ownership analysis was still practical, running within 12 minutes on all benchmarks except one benchmark which took about 44 minutes. The immutability analysis ran in less than 5 minutes on all but two benchmarks, which took about 10 to 15 minutes. The flow analysis including confidentiality and integrity ran within 8 minutes on all benchmarks. The experiments indicated that the analyses scale to large applications.

8.1.2 Applications of The Framework

We have illustrated the usage of our analysis framework on several applications. We have applied ownership inference on reasoning about shared objects in open concurrent Java programs— that is, a library of Java classes designed for use by multi-threaded clients, to facilitate the understanding of concurrent programs. We have proposed three structural patterns for object sharing and conjecture that well-structured concurrent programs should emphasize the role of ownership — they should strive for a larger number of owned objects and for a minimal number of distributed objects. We have performed experiments on several medium-to-large Java programs, which revealed the structure of sharing in real-world Java programs: on average 38% of shared objects were directly access by the client thread, and on average 27% of shared objects were owned by some non-client object.

We have applied the information flow inference on three client applications. The first application was security violation detection. We have performed experiments on a suite of Java web applications and examined whether the applications could be attacked by the injected malicious data. The second application was type inference. We have performed experiments on the same set of Java web applications and inferred the variables "tainted" by information flow from the user injections. The last application was a study of the effect of thread-shared variables on thread-local variables. We have performed experiments on a set of multi-thread programs and studied how many thread-local variables were affected by thread-shared variables. The experiments showed that the information flow analysis had effectively

detected 26 security violations, had automatically inferred thousands of security types, and had illustrated how thread-local variables were affected by thread-shared variables.

On these three applications, we also studied the impact of explicit flow versus implicit flow, showing that implicit flow had significant impact on all these applications.

We believe that our work is a step forward towards the use of static analysis for the purposes of reasoning about program security; it may help advance the use of static analysis in tools for understanding and verification of security properties.

8.2 Future Work

One direction of future work is to extend the general framework in the setting of distributed systems. Unlike centralized control in local systems, the data and service integration in distributed systems make interaction more complex and service or control decentralized, which introduces new problems and challenges. It is necessary to extend localized security properties to distributed specified properties, in order to deal with features in distributed systems. The runtime models are needed to be modified, to capture the expected security-related program behavior in the distributed software infrastructures. Thus the framework should enforce and verify the distributed specified properties, and should give informative results describing the software behavior with respect to these properties.

Another possible direction of future work is to investigate other security-related properties of the framework. For example, the ownership property has two well-known protocols: owner-as-dominator and owner-as-modifier. We are working on the formulation and implementation of static analysis for ownership inference according to the owner-as-modifier protocol. The results could show how frequently the two ownerships occurs in Java programs, and how the owner-as-dominator and the owner-as-modifier protocols capture distinct ownership properties.

We also plan to extend our framework to offer a combination of both static analysis and dynamic analysis, best utilizing the advantages of each methodology while minimizing their disadvantages.

In the future we plan to investigate more applications of our framework. One application we are working on is analysis for data race detection in multi-thread programs. In particular, we base the data race analysis on the object sharing patterns that are described in Section 6.1.3.1. As our study on object sharing indicates that lots of objects are owned by a few central objects, we believe that the ownership analysis and read-only analysis could help detect data races more quickly. Ownership analysis could also help effectively identify of the causes of data races, by showing the data sharing structure relating to the detected data races. Another interesting possibility for future work is the detection and prevention of deadlocks. We believe that this can be achieved by identifying deadlock conditions and capturing the conditions with several analyses.

BIBLIOGRAPHY

[1] Jdk 1.1.1 signing flaw. http://java.sun.com/security/getsigners.html (date last accessed: February 22, 2010).

[2] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[3] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.

[4] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[5] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2002.

[6] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[7] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE TSE*, 30(6):372–386, June 2004.

[8] A. Rountev. Precise identification of side-effect free methods. In *ICSM*, pages 82–91, 2004.

[9] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *ASE*, pages 76–85, 2005.

[10] Samir Genaim and Fausto Spoto. Information flow analysis for Java bytecode. In *VMCAI*, pages 346–362, 2005.

[11] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, 2002.

[12] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 43–55, 2001.

[13] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

[14] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, March 2001.

[15] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–12, 2002.

[16] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[17] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[18] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, LNCS 1781, pages 18–34, 2000.

[19] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[20] Ashes suite collection. http://www.sable.mcgill.ca/software (date last accessed: February 22, 2010).

[21] Dacapo benchmark suite. http://dacapobench.org (date last accessed: February 22, 2010).

[22] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[23] A. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[24] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Simposium*, pages 271–286, 2005.

[25] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[26] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[27] M. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[28] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[29] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.

[30] K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.

[31] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE*, pages 194–203, 2007.

[32] W. Dietl and P. Müller. Runtime universe type inference. In *IWACO*, 2007.

[33] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 301–314, 2004.

[34] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[35] S. Artzi, A. Kieżun, D. Glasser, and M. Ernst. Combined static and dynamic mutability analysis. In *ASE*, pages 104–113, 2007.

[36] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[37] P. Lam and M. Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *ECOOP*, pages 275–302, 2003.

[38] G. Kniesel and D. Theisen. JAC-access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[39] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *In Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, 2002.

[40] D. Volpano and G. Smith. A type-based approach to program security. In *International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, 1997.

[41] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object oriented systems. In *IEEE Symposium on Security and Privacy*, page 130, 1997.

[42] V. Simonet. Flow caml in a nutshell. In *Applied Semantics II Workshop*, pages 152–165, 2003.

[43] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *IEEE Workshop on Computer Security Foundations*, page 16, 2006.

[44] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 241–253, 2001.

[45] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment time confinement checking. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 374–387, 2003.

[46] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[47] R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, pages 149–160, 2004.

[48] Derek Rayside, Lucy Mendel, Robert Seater, and Daniel Jackson. An analysis and visualization for revealing object sharing. In *Workshop on Eclipse technology eXchange*, pages 11–15, 2005.

[49] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, 2000.

[50] M. Lam, M. Martin, B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, 2008.

[51] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 87–97, 2009.

[52] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language design and Implementation*, pages 35–46, 1990.

[53] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34, 2008.

[54] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, pages 87–96, 2006.

[55] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[56] A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *IEEE Computer Security Foundations Workshop*, pages 155–169, 2003.

[57] Q. Sun, A. Banerjee, and D. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Static Analysis Symposium*, pages 84–99, 2004.

[58] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *ACM Network and Distributed System Security Symposium*, 2005.

[59] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[60] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[61] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of USENIX Security Symposium*, pages 191–206, 2002.

[62] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307, 2005.

[63] W. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *ACM International Symposium on Foundations of Software Engineering*, pages 175–185, 2006.

[64] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole-system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[65] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August. Rifle: An architectural framework for user-centric information-flow security. In *IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, 2004.

[66] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *International Workshop on Programming Language Interference and Dependence*, 2005.

[67] S. McCamant and M. Ernst. Quantitative information flow as network flow capacity. In *ACM Conference on Programming Language Design and Implementation*, 2008.

[68] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[69] P. Pratikakis, J. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *ACM Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[70] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM TOPLAS*, 22(1):162–186, 2000.

APPENDIX A
Example Code

```
    public class Register {
      private ProductCatalog catalog;
      private Sale sale;
      public Register() {
1         catalog = new ProductCatalog();                    // o_ProductCatalog
      }
      public void enterItem(ItemId id, int q) {
2         ProductSpec spec = catalog.getSpec(id);
3         sale.makeLineItem(spec, q);
      }
      public void makeNewSale() {
4         sale = new Sale();                                 // o_Sale
      }
      public void makePayment(Money cash) {
5         sale.makePayment(cash);
6         Money balance = sale.getBalance();
      }
      public void endSale() {
7         sale.becomeComplete();
      }
    }

    class ProductCatalog {
8     private Hashtable specs = new Hashtable();             // o_Hashtable
      ProductCatalog() {
9         ItemID id = new ItemID(100);                       // o_ItemID1
10        Money price = new Money(3);                        // o_Money1
```

```
        ProductSpec ps;
11      ps = new ProductSpec(id,price,"TheItem");    // o_ProductSpec
12      specs.put(id,ps);
    }
    ProductSpec getSpec(ItemID id) {
13      return (ProductSpec) specs.get(id);
    }
}

class Sale {
14  private Vector lineItems = new Vector();         // o_Vector
    private Payment payment;
    public Money getBalance() {
15      return payment.getAmount().minus(getTotal());
    }
    public void makeLineItem(ProductSpec s, int q) {
16      lineItems.add(new SaleLineItem(s,q));        // o_SaleLineItem
    }
    public Money getTotal() {
17      Money total = new Money();                   // o_Money2
18      Iterator i = lineItems.iterator();
19      while (i.hasNext()) {
20          SaleLineItem sli = (SaleLineItem) i.next();
21          total.add(sli.getSubtotal());
        }
22      return total;
    }
    public void makePayment(Money cash) {
23      payment = new Payment(cash);                 // o_Payment
    }
    public void becomeComplete() { //log...  }
```

```
}

class SaleLineItem {
  private int quantity;
  private ProductSpec spec;
  public SaleLineItem(ProductSpec s, int q) {
24        this.spec = s; this.quantity = q;
  }
  public Money getSubtotal() {
25        return spec.getPrice().times(quantity);
  }
}

class ProductSpec {
  private ItemID id;
  private Money price;
  private String description;
  public ProductSpec(ItemID id, Money price, String description) {
26        this.id = id; this.price = price; this.description = description;
  }
27  public ItemID getID() { return id; }
28  public Money getPrice() { return price; }
29  public String getDescription() { return description; }
}

class Money {
  private int number;
  public Money(int num) {
        this.number = num;
  }
  public int getNumber() { return number; }
```

```
  public void add(Money other) { number += other.getNumber(); }
  public void minus(Money other) { number -= other.getNumber(); }
  public void times(int factor) { number *= factor; }
}



public class phMain() {
  public static void main() {
30        int q = 0, amount = 0;
31        ItemID id = new ItemID(q);                    // o_ItemID2
32        Money cash = new Money(amount);               // o_Money3
33        Register register = new Register();           // o_Register
34        register.makeNewSale();
35        register.enterItem(id,q);
36        register.makePayment(amount);
37        register.endSale();
  }
}
```

APPENDIX B
Correctness Proof of Information Flow Inference

Consider an arbitrary run-time flow path $s \mapsto \ldots o_1.f_1 \mapsto \ldots o_2.f_2 \mapsto \ldots r$. Our goal is to prove that this path has appropriate analysis representative in $\mathcal{FG}_p$. The path consists of segments $s \mapsto \ldots o_1.f_1$, $o_1.f_1 \mapsto \ldots o_2.f_2$, etc. where all intermediate nodes between dereferences are local variables that are unique for their creating stack frame.

Consider segment $s \mapsto \ldots o_1.f_1$. This segment can have the following structure: $s \mapsto f_1 \overset{ret}{\mapsto} \ldots f_k \overset{call}{\mapsto} \ldots f_n \mapsto o_1.f_1$. Here each f_i represents a stack frame (i.e., a variable-to-variable flow sequence that starts and ends within frame f_i). Edge $f_1 \overset{ret}{\mapsto} f_2$ denotes that frame f_1 returns into frame f_2, and edge $f_k \overset{call}{\mapsto} f_{k+1}$ denotes that frame f_k calls frame f_{k+1}. Furthermore, within each frame f_i there are sequences of balanced frames as follows: $v_1 \overset{call}{\mapsto} g_1 \overset{call}{\mapsto} g_2 \ldots g_{k-1} \overset{call}{\mapsto} g_k \overset{ret}{\mapsto} g_{k_1} \ldots g_2 \overset{ret}{\mapsto} g_1 \overset{ret}{\mapsto} v_2$. Here local variable v_1 in frame f_i flows through a call into stack frame g_1, then there is a sequence of flows within g_1 that end in a call into stack frame g_2, etc. until some frame g_k called from g_{k-1} returns back into g_{k-1}, etc., until finally, g_1 returns back into variable v_2 in f_i. Without loss of generality we may assume that the flows within each g_i are variable-to-variable intraprocedural flows. Our goal is to show that segment $s \mapsto f_1 \overset{ret}{\mapsto} \ldots f_k \overset{call}{\mapsto} \ldots f_n \mapsto o_1.f_1$ has appropriate representative in $\mathcal{FG}^*$. We have the following lemmas:

Lemma 1. For each sequence of balanced frames $v_1 \overset{call}{\mapsto} g_1 \overset{call}{\mapsto} g_2 \ldots g_{k-1} \overset{call}{\mapsto} g_k \overset{ret}{\mapsto} g_{k-1} \ldots g_2 \overset{ret}{\mapsto} g_1 \overset{ret}{\mapsto} v_2$ there is a representative edge $v_1 \rightsquigarrow v_2$ in $\mathcal{FG}^*$.

Sketch of proof. The proof is by induction on the depth of the stack from v_1. For depth of 1 we have path $v_1 \overset{call}{\mapsto} g_1 \overset{ret}{\mapsto} v_2$ where g_1 is a sequence of variable-to-variable intraprocedural flows. Therefore the flow sequence has the form: $v_1 \overset{call}{\mapsto} p \mapsto w_1 \mapsto w_2 \ldots w_n \mapsto ret \overset{ret}{\mapsto} v_2$ where p is the instance of the formal parameter, *ret* is the instance of the return variable, and w_i are the instances of the local variables for stack frame g_1. This path has analysis representative $v_1 \overset{(_1}{\rightsquigarrow} p \rightsquigarrow w_1 \rightsquigarrow w_2 \rightsquigarrow \ldots w_n \rightsquigarrow ret \overset{)_1}{\rightsquigarrow} v_2$, where 1 is the index of the call site that triggers the invocation

of stack frame g_1. It is easy to see that annotation $(_1$ will be propagated through variables p and w_i (lines 3-5 in *Summarize*), resulting in an edge $v_1 \overset{(_1}{\rightsquigarrow} ret$, which will be concatenated with $ret \overset{)_1}{\rightsquigarrow} v_2$ resulting in the needed edge $v_1 \rightsquigarrow v_2$.

Now assume that the lemma is true for depth of k. Let us have a sequence $v_1 \overset{call}{\mapsto} g_1 \mapsto w_1 \overset{call}{\mapsto} g_2 \dots g_2 \overset{ret}{\mapsto} w_2 \mapsto g_1 \overset{ret}{\mapsto} v_2$, where w_1 and w_2 are local variables in frame g_1, and the depth of the stack from w_1 is k. By the inductive hypothesis there is analysis edge $w_1 \rightsquigarrow w_2$ and therefore there is analysis sequence $v_1 \overset{(_1}{\rightsquigarrow} g_1 \rightsquigarrow w_1 \rightsquigarrow w_2 \rightsquigarrow g_1 \overset{)_1}{\rightsquigarrow} v_2$. We need to show that there is analysis edge $v_1 \rightsquigarrow v_2$. Clearly, the analysis adds edge $v_1 \overset{(_1}{\rightsquigarrow} w_1$ due to, possibly multiple, applications of lines 3-5. If this edge is processed on the worklist after edge $w_1 \rightsquigarrow w_2$, *Summarize* will add edge $v_1 \overset{(_1}{\rightsquigarrow} w_2$ and later $v_1 \rightsquigarrow v_2$ due to lines 3-5. Otherwise (i.e., if $v_1 \rightsquigarrow w_1$ is processed before $w_1 \rightsquigarrow w_2$), *Summarize* will add edge $v_1 \overset{(_1}{\rightsquigarrow} w_2$ due to lines 6-8.

Lemma 2. For each sequence of returns $s \mapsto f_1 \overset{ret}{\mapsto} f_2 ... f_k \overset{ret}{\mapsto} v$ there is a representative path edge $s \overset{nCall}{\rightsquigarrow} v$ in $\mathcal{FG}_p$.

Sketch of proof. Let f_i be any frame. The sequence $v_i \mapsto \dots ret$ in f_i (i.e., the sequence that starts in a local v_i in stack frame f_i and ends on return variable *ret* in f_i) has a representative sequence of empty edges $v_i \rightsquigarrow \dots ret$ in $\mathcal{FG}^*$—if the flow between some intermediate pair $w_1, w_2 \in f_i$ is interprocedural, then by Lemma 1 we have an empty representative edge $w_1 \rightsquigarrow w_2$. Furthermore, each return edge from stack frame f_i into stack frame f_{i+1}, namely $ret \overset{ret}{\mapsto} l$, has a representative, namely $ret \overset{)_i}{\rightsquigarrow} l$, where i is the call site that triggers the invocation of stack frame f_i. It is easy to show that procedure *Propagate* computes path edge $s \overset{nCall}{\rightsquigarrow} v$.

Lemma 3. For each sequence of calls $v \overset{call}{\mapsto} f_k \dots f_n \mapsto o_1.f_1$ there are representative edges $v \overset{*}{\rightsquigarrow} p_1.f_1$ in $\mathcal{FG}^*$ for each variable p_1 that points to o and field f_1 is read through p_1.

Sketch of proof. Clearly, for each frame f_i the sequence $p_i \mapsto \dots v_i$ in f_i is represented by an empty-edge sequence in $\mathcal{FG}^*$. Therefore we have sequence $v \overset{(_k}{\rightsquigarrow} p_k \rightsquigarrow v_k \overset{(_{k+1}}{\rightsquigarrow} p_{k+1} \dots p_n \rightsquigarrow v_n$ which will result in $v \overset{(_k}{\rightsquigarrow} v_k \overset{(_{k+1}}{\rightsquigarrow} v_{k+1} \dots v_n$. Without loss of generality we may assume that v_n is the local that reads the value $o_1.f_1$ (i.e., we have a statement $v_n = q.f$ and q points to o_1). Recall that in this case the analysis creates a wildcard edge $v_n \overset{*}{\rightsquigarrow} p_1.f_1$ for each p_1 that points to o_1 and field f_1

is read through p_1. Therefore, we will have sequence $v \overset{(_k}{\rightsquigarrow} v_k \overset{(_{k+1}}{\rightsquigarrow} v_{k+1} \ldots v_n \overset{*}{\rightsquigarrow} p_1.f_1$. Procedure *Summarize* processes this sequence and the processing results in the necessary edge $v \overset{*}{\rightsquigarrow} p_1.f_1$.

Let us return to segment $s \mapsto f_1 \overset{ret}{\mapsto} \ldots f_k \overset{call}{\mapsto} \ldots f_n \mapsto o_1.f_1$. As a result of the above lemmas, this segment will have analysis representative(s) $s \overset{nCall}{\rightsquigarrow} p.f_1$ in $\mathcal{FG}_p$. One can show that if the complete sequence $s \mapsto \ldots o_1.f_1 \mapsto \ldots o_2.f_2 \mapsto \ldots r$ ends on a sequence of calls, it will have analysis representative $s \overset{Call}{\rightsquigarrow} r$ in $\mathcal{FG}_p$; otherwise, it will have representative $s \overset{nCall}{\rightsquigarrow} r$.