PRACTICAL SPECIFICATION AND VERIFICATION OF SECURITY PROPERTIES USING STATIC ANALYSIS

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

August 2008
(For Graduation May 2009)

© Copyright 2008
by
Yin Liu
All Rights Reserved

CONTENTS

LIST OF TABLES . . . v

LIST OF FIGURES . . . vi

ACKNOWLEDGMENT . . . viii

ABSTRACT . . . ix

1. Introduction . . . 1
   - 1.1 Contributions . . . 3
     - 1.1.1 Security Specification as Annotations on UML diagrams . . . 3
     - 1.1.2 Security Verification with Runtime Models and Static Analyses 4
     - 1.1.3 Practical and Precise Analyses Results . . . 5
   - 1.2 Thesis Outline . . . 6
2. Overview And Motivating Example . . . 7
   - 2.1 POS System Description . . . 8
   - 2.2 POS System Security Specification . . . 8
   - 2.3 POS System Security Verification . . . 9
     - 2.3.1 Runtime Models . . . 9
     - 2.3.2 Static Analyses . . . 10
3. Framework Setting . . . 12
   - 3.1 The Case for UML-based Specification and Verification . . . 12
   - 3.2 Problem Statement . . . 13
     - 3.2.1 Whole Program . . . 13
     - 3.2.2 Software Components . . . 14
4. Runtime Models . . . 16
   - 4.1 Ownership Model . . . 16
   - 4.2 Immutability Model . . . 18
   - 4.3 Information Flow Model . . . 19

5. Static Analysis Framework . . . 26
5.1 Fragment Analysis . . . 26
5.2 Points-to Analysis . . . 27
5.3 Ownership Analysis . . . 28
5.3.1 Approximate Object Graph . . . 28
5.3.2 Ownership Inference . . . 30
5.3.3 Improved Ownership Inference . . . 32
5.3.4 Correctness proof and Complexity . . . 33
5.4 Immutability Analysis . . . 35
5.4.1 Immutability Inference . . . 35
5.4.2 Improved Immutability Inference . . . 36
5.4.3 Complexity . . . 37
5.5 Information Flow Analysis . . . 38
5.5.1 Context-insensitive Information Flow Analysis . . . 38
5.5.2 Implicit flow . . . 41
5.5.3 Context-sensitive Shallow Flow Analysis . . . 42
5.5.3.1 Imprecision of Context-insensitive Analysis . . . 42
5.5.3.2 Construction of Flow Graph $\mathcal{FG}_0$ . . . 43
5.5.3.3 Summarization . . . 45
5.5.3.4 Propagation . . . 47
5.5.3.5 Termination, Complexity and Correctness . . . 49
5.5.4 Deep Flow Analysis . . . 50
5.5.5 Confidentiality Inference . . . 51
5.5.6 Integrity Inference . . . 52
6. Empirical Results . . . 56
6.1 Software Components . . . 57
6.1.1 Analysis Precision . . . 57
6.1.2 Analysis Cost . . . 58
6.2 Complete Programs . . . 59
6.2.1 Analysis Precision . . . 62
6.2.2 Analysis Cost . . . 63
6.3 Conclusions . . . 64

7. Related Work 65

7.1 Type systems 66

7.2 Ownership inference 66

7.3 Immutability inference 67

7.4 Static information flow analysis 68

7.5 Dynamic information flow tainting 68

7.6 CFL-reachability 69

7.7 Our approach 69

8. Conclusions and Future Work 70

BIBLIOGRAPHY 71

APPENDICES

A. Example Code 76

LIST OF TABLES

6.1 Information of Java components. 57

6.2 Owned fields. 58

6.3 Immutable fields, methods and parameters. 58

6.4 Confidentiality (fields leaked to client code) and integrity (fields tempered by client code). 59

6.5 Analysis time. 59

6.6 Information about the Java benchmarks. 60

6.7 Owned fields. 61

6.8 Immutable fields, methods and parameters. 61

6.9 Columns 3 and 4: Confidentiality; Columns 5 and 6: Integrity. 62

6.10 Analysis time. 63

LIST OF FIGURES

2.1 UML class diagram with ownership, immutability and information flow constraints. 7

4.1 Object graph for Figure 4.2. 17

4.2 Simplified vector and its iterator. 23

4.3 Sample package `zip`. 24

4.4 Placeholder `main` method for `zip`. 25

5.1 Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty. 29

5.2 Partial Ag for Appendix A. 30

5.3 Ownership inference: computing the closure of edge $h_i \rightarrow h_j$. $Tgts(h)$ stands for $\{h' \mid h \rightarrow h' \in Ag\}$ and $Srcs(h)$ stands for $\{h' \mid h' \rightarrow h \in Ag\}$. 31

5.4 Immutability inference: computing the read-only status of $h_i \rightarrow h_j$. . . . 35

5.5 Imprecision of immutability inference. 37

5.6 Computation of conditional range. 54

5.7 Computation of shallow flow from s. 55

5.8 Computation of deep flow. 55

ACKNOWLEDGMENT

ABSTRACT

We present a model-based approach for specification and verification of security-related program properties. Our approach advocates practical specification using UML class diagrams and practical verification using reverse engineering based on scalable static analysis.

Specifically, we propose the use of ownership, immutability, and information flow constraints on UML class diagrams, and verification of these constraints through reverse engineering.

At the heart of the approach is scalable static analysis. We present ownership, immutability and information flow inference analyses for Java. These analyses are practical and precise and therefore can help support effective reasoning about software security and software quality.

We perform an empirical investigation on several Java components, and a set of small-to-large Java programs. The results indicate that the inference analyses are precise and practical. Therefore, the analyses can be integrated in reverse engineering tools and can help support effective reasoning about software quality and security.

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 specifying and verifying 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.

It is important to study mechanisms for specifying and verifying 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 changes to the language and runtime as well as non-trivial type annotations; thus, it might be difficult to adopt these approaches in practice.

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

We propose a new general-purpose framework based on static analysis, for specification and verification of security-related program properties. We advocate reasoning based on the universally-known Unified Modeling Language (UML) class diagrams. Developers can identify important constraints during design, and *choose* to specify these constraints on their design class diagrams, exactly where needed. They can later check, verify, and reason about their code as needed: violations of specified constraints is revealed intuitively in the reverse-engineered UML class diagram. We believe that this practical approach to specification and verification can help realize the benefits of reasoning about ownership, immutability and information flow in real-world software engineering practice.

As mentioned earlier, our approach allows specification of security properties as the constraints on UML class diagrams. We specify object access and information flow properties as *ownership*, *immutability* and *information flow* constraints. Note that these are the constraints we have explored so far; the framework can be easily extended with other useful constraints.

These constraints force reasoning about security properties–specifically,object access and information flow–at design level, and impose requirements on the program runtime behaviors. These runtime behavior requirements are illustrated by the runtime models, which specify what does the design level security constraints mean at runtime. The constraints can be continually verified through reverse engineering via applying static analysis directly on code, which retrieves the implemented constraints according to the runtime models.

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 security specification and verification framework 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.

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 Security Specification as Annotations on UML diagrams

We present a practical general-purpose framework for specification and verification of security properties. The security properties are captured by annotations on UML diagrams.

The UML diagram annotations make security property more understandable. They also inherently support software design principles: e.g. the ownership and immutability constraints support "Low Coupling" and "Information Expert" [1]. Developers can identify important constraints during design, and *choose* to specify these constraints on their design class diagrams, exactly where needed. They can later check, verify, and reason about their code as needed: violations of specified constraints, or other unintented access or unintended information flow is revealed intuitively in the reverse-engineered UML class diagram. We believe that this practical approach to specification and verification can help realize the benefits of reasoning about ownership, immutability and information flow in real-world software engineering practice.

Specifically, the security constraints we have explored are *ownership*, *immutability* and *information flow*.

The security property of object access control is 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 implemen-

tation 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 of data confidentiality and data integrity are specified 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.

1.1.2 Security Verification with Runtime Models and Static Analyses

Our framework retrieves and verifies security properties with runtime models and static analyses. The framework could be augmented with different runtime models and accordingly 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 objectthat 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 runtime models, we propose novel static analyses that infers ownership, immutability, and information flows directly from the code, according to these models. The inference is based on points-to analysis, which determines the set of objects a reference variable or a reference object field may point to.

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 on complete programs. Thus this approach can be easily incorporated in program understanding and verification tools and help verify in a lightweight and practical manner the program security properties.

1.1.3 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 average 28% of the reverse-engineered associations were determined to be `owned`,

27% were determined to be `read-only`, and 43% of the examined data were determined to be `deep leaked`, 24% were determined to be `deep tampered`. We present a precision evaluation which indicates that the analyses achieve adequate precision—the ownership inference almost never misses an `owned` association, the immutability inference rarely misses a `read-only` association, and almost all identified confidentiality and integrity violations could actually happen. The ownership and immutability analyses are practical, running in less than 7 minutes on all but one benchmark. The flow analysis including confidentiality and integrity runs within 8mins on all benchmarks. The experience indicates 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.

1.2 Thesis Outline

The rest of this thesis is organized as follows. The general idea of our framework for security specification and verification is presented in Chapter 2, illustrated with a motivating example. Chapter 3 describes the problem setting of our framework. Chapter 4 defines the runtime models that capture the design level security properties of *ownership*, *immutability* and *information flow*. Chapter 5 outlines the framework, and presents the ownership, immutability and information flow analyses. Chapter 6 discusses the experimental study. 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 specification and verification of security properties could be generalized as: clarify the security requirement of the design, and specify the security property on the design UML diagrams; define run-time requirements as run-time models that capture these design level security properties; then verify the security properties by applying static analysis on code, which analyze the run-time behavior of the program without running the code.

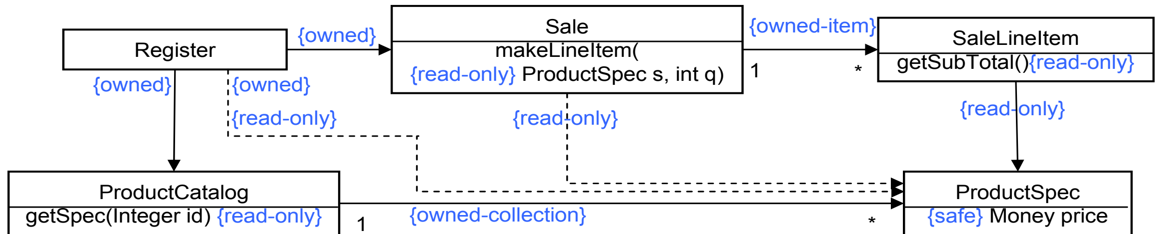

Figure 2.1: UML class diagram with ownership, immutability and information flow constraints.

To illustrate the general idea of our framework, consider a component of a Point-of-Sale (POS) system [1], 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 formalize the design and security requirements, which force verification of these requirements on the implementation. Specifying these constraints on UML class diagrams, and verifying them through reverse engineering can help reasoning about security-related properties. It may lead to higher quality, more secure and understandable software.

2.1 POS System Description

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.

2.2 POS System Security Specification

Figure 2.1 presents a UML class diagram for the POS system which is annotated appropriately.

The constraints that `Register` owns all objects it accesses (i.e., `ProductCatalog`, `Sale` and `ProductSpec`) specify the security requirement that these objects are part of the internal representation of the system and should not be leaked outside (i.e., to potentially untrusted clients).

The constraint that `SaleLineItem` has read-only access to `ProductSpec` forbids `SaleLineItem` from modifying `ProductSpec`, and the constraint that method `makeLineItem` in class `Sale` has read-only access to the parameter `s` (of type `ProductSpec`) forbids `makeLineItem` from modifying the `ProductSpec` object referred by `s`. This is because the `ProductCatalog` is the "information expert" and the only object that can initialize and update product information. The constraints that methods `getSpec` in class `ProductCatalog` and `getSubtotal` in class `SaleLinItem`

are read-only, forbid these methods from modifying certain visible state.

Finally, the constraint that field *price* (of type `Money`) in `ProductSpec` is `safe`, specifies an integrity requirement: 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 `ProductSpec` objects are owned by the `Register` object; however, the ownership constraint does not prevent deeper information flow violations, such as leaks or modifications to sensitive data which is part of the owned `ProductSpec` objects.

2.3 POS System Security Verification

The security properties specified on UML diagrams impose requirements on the program runtime behaviors. It is needed to define runtime models that illustrate runtime behavior requirements to capture the design level constraints. Then static analysis could be utilized to analyze the code according to the runtime models.

2.3.1 Runtime Models

The runtime models are defined to specify runtime behavior requirements according to the security constraints.

The ownership model is based on the notion of *owners-as-dominators* [2, 5, 3]. Essentially, as 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 modified object sets by an method execution of $o.m$. As in Figure 2.1, the constraint that the parameter `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 `SaleLinItem.getSubtotal` are read-only, forbid the runtime invocation of these methods from modifying any of their own parameters and any static fields. The

constraint that `SaleLineItem` has read-only access to `ProductSpec` requires that any runtime method invocation on object $o_{SaleLineItem}$ should not modify any part of its field object $o_{ProductSpec}$.

The information flow model captures both explicit, implicit, and both shallow, deep flows. 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 malicious part to the object o_{Money} of the field *ProductSpec.price*.

2.3.2 Static Analyses

With the runtime models which illustrate runtime behavior requirements according to the design constraints, the static analyses analyze the code, reason about its runtime behaviors and reverse engineer the implemented constraints.

For completeness, we have included the Java code from [1, 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 runtime 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 analyzed the code in Appendix A, and concludes that all possible invocation of $o_{Sale}.makeLineItem$ does not modify the transitive object state of the parameter `s`, and all possible invocation of method `getSpec` in class `ProductCatalog` does not modify any object state observable from outside of method call. 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 an modification of the field

SaleLineItem.ProductSpec by method invocation of `SaleLinItem.getSubtotal`. Thus method `SaleLinItem.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. As a result, subsequent sales fetched `ProductSpec`s with wrong prices and computed incorrect sale totals!

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

CHAPTER 3
Framework Setting

Our framework supports understandable specification and light-weight verification of security properties, through scalable, informative and widely-used UML diagrams. It could specify and verify various security properties, while augmented with corresponding analyzes. To retrieve or verify each type of security property constraint, it is necessary to define the security problem, within the setting of complete programs and within the setting of software components.

3.1 The Case for UML-based Specification and Verification

It is well-known that reasoning about program security properties such as ownership, immutability and information flow has important software engineering benefits. Such reasoning would not only help prevent serious security flaws, but would lead to higher quality, more understandable and maintainable software systems.

Consequently, there are numerous langauge-based proposals that address these issues—there are proposals for language extensions with ownership types [2, 6, 7, 5], immutability types [8] and information flow types [9]. However, these approaches require a large amount of code annotations, and writing code annotations requires significant effort. Many of the needed annotations are far away located in code and seemingly irrelevant to the property of interest; furthermore, annotations are written in an unfamiliar annotation langauge which may be difficult to master. The burden is significant and could make type-based approaches unlikely in practice.

We propose a practical, model-based approach to reasoning about ownership, immutability and information flow, and consequently, reasoning about important security properties. As mentioned earlier, we advocate reasoning based on the universally-known UML class diagrams, which makes security property more understandable. Developers can identify important constraints during design, and *choose* to specify these constraints on their design class diagrams, exactly where needed.

They can later check, verify, and reason about their code as needed: violations of specified constraints, or other unintented access or unintended information flow is revealed intuitively in the reverse-engineered UML class diagram. We believe that this practical approach to specification and verification can help realize the benefits of reasoning about ownership, immutability and information flow in real-world software engineering practice.

The key to making this approach work, is the development of analysis that could reverse-engineer constraints for UML class diagrams precisely and efficiently. The rest of the paper describes the design of a static analysis framework for this purpose, and instantiations with analyses that infer ownership, immutability and information flow constraints.

3.2 Problem Statement

The framework works not only on complete programs (i.e., whole programs) , but 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 software components and thus cannot accommodate analysis that works on 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.

3.2.1 Whole Program

Conventionally, software tools reverse engineer UML associations by examining associations or elements that are annotated with security related constraints.

For `owned` and `read-only` constraints on associations, our framework examines instance fields of reference type in the code (e.g., a field f of type B in class A is reverse engineered into an association from A to B labeled with f).[1] The ownership

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

inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, o owns o'. Similarly, the immutability inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, o does not mutate o'.

For `read-only` constraints on parameters, the immutability inference problem is to find the parameters p_i of method m such that for all run-time objects o_{p_i}, invocations of m does not mutate o_{p_i}. For `read-only` constraints on methods, the immutability inference problem is to find the method m such that for all run-time invocations of $o.m$, no parameter of m nor any static field reachable before method invocation is mutated.

For `confidential` and `safe` constraints on fields, our framework examines all instance fields in the code. The confidentiality inference problem is to find the fields f such that for each run-time object $o.f$, the object state of untrusted destinations, *sinks*, such as OutputStreams. Similarly, the integrity inference problem is to find the fields f such that for each run-time $o.f$, the object state of $o.f$ is not even partially effected by any flow from untrusted sources, *sources*, such as InputStreams.

3.2.2 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. 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. The classes in *Cls* are trusted while the client code built on top of these classes is untrusted.

Consider the code in Appendix A which implements the POS system design from Figure 2.1 in Chapter 2; it is taken from [1][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 consider ownership, immutability and information flow constraints as outlined in the previous sections. There are two issues. First, we need to define run-time models for these constraints (e.g., what does it mean precisely that a run-time

`Register` object owns the run-time `Sale` object that it references? Similarly, what does it mean that field *price* is `safe`?). Second, we need to design static analyses that answer the following questions: given a constraint c, does there exist a client of *Cls* that violates c (e.g., does there exist a client such that some `Register` object does not own its `Sale` object? Similarly, does there exist a client that modifies the price of some product?).

Problem Boundary

We employ the following constraint, which is standard for other problem definitions that require analysis of incomplete programs [10, 11, 12]. We only consider executions in which the invocation of a boundary method 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 6 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 this case the analysis will be overly conservative and will likely report less useful information.

CHAPTER 4
Runtime Models

To verify security properties specified by constraints, we need run-time models to define requirements on program runtime behaviors imposed by the constraints. Particularly, we consider ownership, immutability and information flow constraints as outlined in the previous section. 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.

4.1 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [2, 5, 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 $o \rightarrow o'$ from run-time object o to run-time object o' (i.e., we say that o' is accessed in the *context* of o) if and only if one of the following is true:

- Reference instance field f in o refers to o'.
- Object o is an array object with element o'.
- An instance method or constructor invoked on receiver o has local variable r that refers to o', or a static method called from an instance method or constructor invoked on o, has a local variable r that refers to o'.[2]

[2]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.

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.

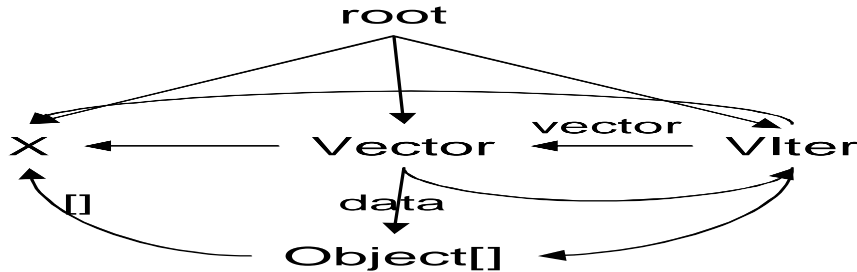

Figure 4.1: Object graph for Figure 4.2.

In accordance with the owners-as-dominators model, we say that o *owns* o' if and only if o is the immediate dominator of o' in the object graph at all execution points.[3] Consider the partial object graph in Figure 4.1; it is a summary graph, which includes all object graphs during the execution of `main` in Figure 4.2. Node `root` represents the start of program execution. The other nodes correspond to the objects created at the appropriate allocation sites in Figure 4.2. `Vector` does not own `Object[]` because during the execution of `next` there is a temporary access from `VIter` to `Object[]`. `Vector` would own `Object[]` if `next` is never executed (i.e., line 15 is 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].

The ownership inference problem for complete programs therefore is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, o owns o' 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, each run-time edge $o \xrightarrow{f} o'$, o

[3] Node m *dominates* node n if every path from the root of the graph that reaches node n has to pass 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.

owns o' according to the above definition. The associations through these fields are marked as `owned`.

4.2 Immutability Model

Let e be an execution of a method m on receiver object o; e modifies an object o' if it triggers a change in the object structure rooted at o'—that is, e leads to a statement $p.f = q$ which writes some o'' reachable from o' (i.e., p refers to o'' and there is a path of field edges from o' to o''). For example, the execution of `add` with receiver `Vector` (line 13 in Figure 4.2) modifies `Object[]`.

A parameter p_i of method $m(...p_i...)$ is `read-only` if no execution of method m modifies the object o referred by p_i. Consider the execution of method `add` with run-time receiver `Vector` from Figure 4.2. The field `data` of `Vector` is reset by the method execution, which is partial object sate of `Vector`. Thus the *this* parameter of method `add`, the receiver object `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 referred by a static field. As in previous example, method `add` is not read-only, as its *this* parameter is modified by its execution. [4]

Finally, we say that *o has read-only access to o'* if no execution of a method on receiver o modifies o'. An association from class A to class B is `read-only`, if and only if for every client of *Cls*, each instance of A has read-only access to the corresponding instances of B. Consider run-time edge `Vector` $\stackrel{data}{\rightarrow}$ `Object[]` from Figures 4.2 and 4.1. Clearly, `Vector` does not have read-only access to array `Object[]` because the execution of method `addElement` with receiver `Vector` (line 13) modifies `Object[]`. Note that the model considers *transitive* modifications (i.e., an execution may modify fields of o' as well as fields of fields of o', and so on.). As another example, consider edge `VIter` $\stackrel{vector}{\rightarrow}$ `Vector`. `VIter` has read-only access to `Vector` because the execution of `nextElement` with receiver `VIter` (line 15) does

[4] 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.

not modify the structure rooted at `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 $o \xrightarrow{f} o'$ o has read-only access to o'. 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 $o \xrightarrow{f} o'$ o has read-only access to o'. The associations through these fields are marked as `read-only`.

4.3 Information Flow Model

Intuitively, there is information flow from variable x into variable y, denoted by $x \mapsto y$ if changes in the input values of x are observable from the output values of y. Such flows are *direct* and *indirect* [4, 13]. Direct flows can be *explicit* (i.e., data-flow based) and *implicit* (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 $v_i \mapsto l$ s.t. 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 $r \mapsto o.f$, where o is the run-time object referred by l at the point of execution of the statement, and implicit flows $v_i \mapsto o.f$ 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 $o.f \mapsto l$, 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 is the run-time object referred by r at the point of execution of the statement.

- **Method call** $l = r_0.m(r_1, ...)$ dispatched to method $m'(\mathit{this}, p_1, ..., \mathit{ret})$ leads to flows $r_0 \mapsto \mathit{this}$, $r_1 \mapsto p_1$, ... and $\mathit{ret} \mapsto l$, and implicit flows $v_i \mapsto m'$, $v_i \mapsto \mathit{this}$, $v_i \mapsto p_1$, ... $v_i \mapsto l$, 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', 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, 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 $l_1 \mapsto o.f$, and then the execution of $l_4 = l_3.f$ (l_2 and l_4 both point to object o) leads to flow $o.f \mapsto l_4$. 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' which can be reached on a sequence of field dereferences from the object o 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 program 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 expose 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 expose information flow from some variable v in the client code into s?

Example. Consider package `zip` in Figure 4.3. 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. 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.[5] For example, consider the following client:

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

Let o_{ZIS} stand for the run-time `ZipInputStream` object created in the above client. The shallow flow results as follows: `readLOC.e` $\mapsto$ `readLOC.ret` due to line 10, followed by `readLOC.ret` $\mapsto$ `getNextEntry.e` due to line 2, followed by `getNextEntry.e` $\mapsto$ o_{ZIS}`.entry` due to line 3, followed by o_{ZIS}`.entry` $\mapsto$ `getNextEntry.ret` due to line 4, and finally `getNextEntry.ret` $\mapsto$ `ph_ZE` due to the call to `getNextEntry` in the client. Second, consider flow from field `tmpbuf` in `ZipInputStream`. Let o_{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

[5]For the rest of the paper we employ the following notation: `m.v` denotes local variable v in method m, `m.v.f` denotes field f referenced through local variable v in method m, and o`.f` denotes field f of object o.

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 invocation of `getNextEntry` followed by the invocations of `readLOC`, `get32` and `get16` triggers aliasing of `tmpbuf` and `get16.b`. Then there is a dereference `get16.b[off]` at line 11 (which is an alias of `tmpbuf[]`), and shallow flow from `b[off]` into `get16.b1`, `get16.i1`, `get16.ret`, `get32.ret`, `readLOC.i2`, o_{ZE}`.size`, `getSize.ret`, and finally into variable `ph_long`.

```
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 4.2: Simplified vector and its iterator.

```
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 4.3: Sample package zip.

```
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 4.4: Placeholder `main` method for `zip`.

CHAPTER 5
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* [14, 10] which enables analysis of partial programs (Section 5.1). Furthermore, the problems require points-to information and we employ a general-purpose points-to analysis (Section 5.2).

The fragment analysis and the points-to analysis are a general foundation that allows building of different 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 (Chapter 4). 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.

5.1 Fragment Analysis

Clearly, the problem considered in this paper requires analysis of a partial program. 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 [14, 10]. 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* [14, 10].

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 [14, 10].

5.2 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 Andersen's points-to analysis [15, 16]. 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 (i.e., objects are distinguished by their allocation sites).

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; the objects created at an allocation site s_i are represented by object name $h_i \in O$. Set F contains all instance fields in program classes. The analysis solution is a *points-to graph* where the edges represent the following "may-refer-to" relationships.

- Let $r \in R$ and $h \in O$. An edge (r, h) in the points-to graph means that at run time r may refer to some object that is represented by h.

- Let $f \in F$ be a reference instance field in objects represented by some $h \in O$. An edge $(h.f, h_2)$ means that at run time field f of some object represented by h may refer to some object represented by h_2.

- Let h represent the array objects. An edge $(h[], h_2)$ shows that some element of some array represented by h may refer at run time to an object represented

by h_2.

For the rest of the paper we use notation h, h_i, h_j, etc. to denote analysis objects (i.e., the object names corresponding to allocation sites used by our analysis to represent run-time objects). In contrast, we use notation o, o_i, o_j, etc. to denote run-time objects.

Most points-to analyses, including Andersen's points-to analysis, 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 approximates 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 [14, 10].

5.3 Ownership Analysis

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

5.3.1 Approximate Object Graph

The nodes in *Ag* are taken from the set of object names O and the edges represent "may-access" relationships. Figure 5.1 outlines the construction of *Ag* given a points-to graph *Pt*. Intuitively, the algorithm tracks flow of objects from one context to another context. Lines 1-2 account for edges due to object creation; at object allocation sites (i.e., constructor calls), the 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 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 calls, $\mathcal{C}_m$ includes the special context `root` as well. At allocation sites new edges are added to *Ag*

input *Stmt*: set of statements *Pt*: $R \cup O \rightarrow \mathcal{P}(O)$
output $Ag : O \rightarrow \mathcal{P}(O)$
[1] **foreach** statement s in method m
$s_i : l = new\ \ C(...)$
[2] add $\{c \rightarrow h_i \,|\, c \in \mathcal{C}_m\}$ to Ag
//**creation** flow into caller contexts
[3] **foreach** statement s in method m
$s : l = r.n(...)$ s.t. $r \neq$ `this`,
$s : l = r.f$ s.t. $r \neq$ `this`
[4] add $\{c \rightarrow h_j \,|\, c \in \mathcal{C}_m \wedge (l, h_j) \in Pt\}$ to Ag
//**outflow** from callee contexts to caller contexts
[5] **foreach** statement s in method m
$s : l = new\ \ C(r)$,
$s : l.n(r)$ s.t. $l \neq$ `this`,
$s : l.f = r$ s.t. $l \neq$ `this`
[6] add $\{h_i \rightarrow h_j \,|\, (l, h_i) \in Pt \wedge (r, h_j) \in Pt\}$ to Ag
//**inflow** into callee contexts from caller contexts
[7] label with f each $h_i \rightarrow h_j \in Ag$ s.t. $(h_i.f, h_j) \in Pt$

Figure 5.1: Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

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. For example, at an instance call not through `this` new edges are added from each context of m to each returned object. Finally, lines 5-6 account for edges due to flow from the contexts of m into other contexts. For example, at an instance call, edges are added from each object in the points-to set of the receiver to each object in the points-to set of a reference argument. Finally, line 7 labels with field identifier f each edge $h_i \rightarrow h_j \in Ag$ for which there is an edge $(h_i.f, h_j) \in Pt$.

As an example, consider the code in Figure 4.2. In this case, the algorithm in Figure 5.1 constructs precisely the summary run-time object graph in Figure 4.1. Edges `root`→`Vector`, `root`→`X` and `Vector`→`Object[]` are due to code lines 11, 12 and 1 respectively (lines 1-2 in the algorithm). Edge `Vector`→`X` is due to code line 13 (lines 5-6 in the algorithm). Edge `Object[]`→`X` is due to code line 2. Fur-

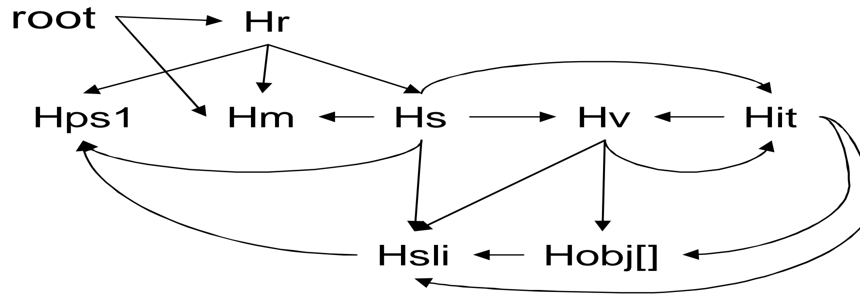

Figure 5.2: Partial *Ag* for Appendix A.

thermore, edge `root`→`VIter` is due to code line 14, and edges `Vector`→`VIter` and `VIter`→`Vector` are due to line 4. Finally, edges `VIter`→`Object[]` and `VIter`→`X` are due respectively to lines 7 and 10 in `next`.

The object graph construction and ownership inference need to consider two special cases: (i) static fields and (ii) self-references (i.e., an object references itself through `this` as in `r.m(this)`). For brevity, we do not discuss these cases; our implementation handles them correctly.

5.3.2 Ownership Inference

The ownership inference uses *Ag* to reason about object ownership. Consider the partial object graph in Figure 5.2, extracted from the code in Appendix A for Chapter 2 from [1]. Node `root` represents the special context of `main` and node *Hr* represents the `Register` object (created in `main`). *Hps1* represents `ProductSpec` objects (created in `ProductCatalog`), *Hm* represents `Money` objects (created in `main` to account for payment for a sale), and *Hs* represents `Sale` objects (created in `Register` when initiating a new sale). *Hsli* represents `SaleLineItem` objects (created in `Sale` when processing a new line item) and *Hv* represents the collection needed to store the `SaleLineItem`s. Finally, *Hit* represents iterators over the collection of `SaleLineItems` (used in `Sale` when calculating the sale total).

The inference analysis (Figure 5.3) examines an edge $h_i \rightarrow h_j$ in the object graph and attempts to prove that for *each* run-time instance $o_i \rightarrow o_j$ of that edge o_i dominates o_j; intuitively, it reasons about the flow of run-time objects based on the object graph abstraction of this flow. The inference is based on the following

input Ag: $O \rightarrow \mathcal{P}(O)$ $h_i \rightarrow h_j$: $O \times O$
output $Closure$: $O \rightarrow \mathcal{P}(O)$, isClosed: boolean
[0] if $isOutside(h_i \rightarrow h_j)$ **return** false
[1] $Closure{=}\{h_i, h_j\}$, $W{=}\{h_i\}$
[2] while W not empty
[3] take h_k from W
[4] foreach $h_m \in Tgts(h_k) \cap Closure$
[5] foreach $h_n \in Tgts(h_k) \cap Srcs(h_m)$, $h_n \notin Closure$
[6] if $isOutside(h_i \rightarrow h_n)$ **return** false
[7] if $valid(h_k, h_n, h_m)$ add h_n to $Closure$ and to W
[8] foreach $h_m \in Srcs(h_k) \cap Closure$
[9] foreach $h_n \in Tgts(h_m) \cap Srcs(h_k)$, $h_n \notin Closure$
[10] if $isOutside(h_i \rightarrow h_n)$ **return** false
[11] if $valid(h_m, h_n, h_k)$ add h_n to $Closure$ and to W
[12] **return** true

procedure $valid$
input h_i, h_k, h_j, where $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$
output isValid: boolean
[1] if $isIn(h_k \rightarrow h_j)$ and $h_i \in In(h_k \rightarrow h_j)$ **return** true
[2] if $isOut(h_i \rightarrow h_j)$ and $h_k \in Out(h_i \rightarrow h_j)$ **return** true
[3] **return** false

Figure 5.3: Ownership inference: computing the closure of edge $h_i \rightarrow h_j$. $Tgts(h)$ stands for $\{h' \mid h \rightarrow h' \in Ag\}$ and $Srcs(h)$ stands for $\{h' \mid h' \rightarrow h \in Ag\}$.

intuition: an object o_j can flow from o_i into some o_k only if one of the following is true: (1) o_k has a handle to both o_i and o_j (and hence Ag contains edges $h_k \rightarrow h_i$, $h_k \rightarrow h_j$ and $h_i \rightarrow h_j$), or (2) o_i has a handle to both h_k and h_j (and hence Ag contains edges $h_i \rightarrow h_k$, $h_i \rightarrow h_j$, $h_k \rightarrow h_j$). In Figure 5.2 edge triple $Hr{\rightarrow}Hps1$, $Hr{\rightarrow}Hs$, $Hs{\rightarrow}Hps1$ represents the fact that a `ProductSpec` object flows into a `Sale` object from the `Register` object. Note that if an edge $h_i \rightarrow h_j$ in Ag does not have an h_k such that either (1) h_k has handles to both h_i and h_j, or (2) h_i has handles to both h_k and h_j, we have that each o_i exclusively owns each o_j it refers to (i.e., o_i is the only object that has a reference to o_j). In Figure 5.2 $Hr{\rightarrow}Hs$ is such an edge; it represents that the `Register` exclusively owns the `Sale` objects it creates.

Consider the algorithm in Figure 5.3, lines 0 to 12, assuming that $valid$ always

returns true; the role of *valid* will be explained shortly. The algorithm makes use of a predicate *isOutside*$(h_i \rightarrow h_j)$ (lines 0, 6 and 10)—an edge $h_i \rightarrow h_j$ is an *outside edge* if there exists an h_k such that h_k has handles to both h_i and h_j. Intuitively, *isOutside* conservatively captures the situation when some o_j flows from (or into) an "outside" object o_k and therefore there might be an access path to o_j that does not pass through o_i. In Figure 5.2, edge *Hs*→*Hm* is an outside edge. The `Money` object is passed from the `Register` to a `Sale` and the `Sale` object does not own it. If the edge that is examined, namely $h_i \rightarrow h_j$, is not an outside edge, the algorithm proceeds to compute the *Closure* of $h_i \rightarrow h_j$. The algorithm finds all paths from h_i to h_j. It examines each edge $h_1 \rightarrow h_2$ in *Closure* and adds nodes h_3 such that there is a triple $h_1 \rightarrow h_2$, $h_1 \rightarrow h_3$ and $h_3 \rightarrow h_2$. If at some point the algorithm detects a path that originates in an outside edge, it returns false (lines 6 and 10). If the algorithm returns true, it is guaranteed that for each edge $o_i \rightarrow o_j$ represented by $h_i \rightarrow h_j$, all paths from o_i to o_j are internal (i.e., o_i dominates, and thus owns o_j). The proof of this statement is given in [17].

Consider edge *Hr*→*Hps1* in Figure 5.2. The algorithm processes h_k equal to *Hr*, *Hs*, *Hsli*, *Hv*, *Hit*, and *Hobj[]*, in this order. It returns true and computes the closure which consists of the above nodes plus *Hps1*. The closure captures all nodes where the `ProductSpec` objects may flow; they are all within the boundary of the `Register`.

If the algorithm in Figure 5.3 returns true for every edge labeled with f, the ownership analysis concludes that the association through f is `owned`.

5.3.3 Improved Ownership Inference

Note that the analysis, as described above will likely incur substantial imprecision and cost. This is due to the fact that not all edge triples $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$ represent valid flow. For example, suppose that edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ are due to object creation (lines 1-2 in the algorithm in Figure 5.1) and $h_i \rightarrow h_k$ is due to inflow (lines 5-6). Clearly, edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ refer to two distinct run-time objects that are represented with the same name, h_j. However, the analysis concludes that there might be an o_j that flows from some o_i into some o_k

and erroneously infers that edge $h_i \to h_j$ is not owned. Invalid triples affect not only precision but cost as well. In the above example, when reasoning about edge $h_i \to h_j$ the analysis needs to reason about edges $h_i \to h_k$ and $h_k \to h_j$, which is clearly redundant as these edges are irrelevant to $h_i \to h_j$.

The edges in the object graph may be characterized as *creation* (due to lines 1-2 in Figure 5.1), *outflow* (due to lines 3-4) and *inflow* (due to lines 5-6). First, let $h_i \to h_j$ be an outflow edge. A triple with h_k (i.e., $h_i \to h_j$, $h_i \to h_k$ and $h_k \to h_j$) will be a valid triple only if for some statement $l = r.n()$ that produces outflow edge $h_i \to h_j$ we have that r point to h_k. Second, let $h_k \to h_j$ be an inflow edge. A triple with h_i (i.e., $h_k \to h_j$, $h_i \to h_k$ and $h_i \to h_j$) will be a valid triple only if for some statement $l.n(r)$ that produces this edge we have that the `this` pointer of the enclosing method of $l.n(r)$, point to h_i.

The algorithm in Figure 5.1 is augmented to track valid sources for outflow and inflow edges. Lines 4' and 6' below are added respectively after lines 4 and 6; there is a set *Out* for each outflow edge and a set *In* for each inflow edge.

[4'] add $Pt(r)$ to $Out(c \to h_j)$

[6'] add $Pt(\texttt{this}_m)$ to $In(h_i \to h_j)$

Subsequently, the ownership inference in Figure 5.3 uses procedure *valid* to filter out invalid triples. For example, if $h_k \to h_j$ is an inflow edge, h_i must appear in $In(h_k \to h_j)$.

5.3.4 Correctness proof and Complexity

Brief correctness proof. Let the algorithm return true for some edge $h_i \to h_j$. Then we have that for each $o_i \to o_j$ represented by $h_i \to h_j$, o_i dominates o_j.

We give the correctness argument for this statement. Let $o_i \to o_j$ be an edge represented by $h_i \to h_j$ and let $\mathcal{K}(o_i \to o_j)$ denote the paths from o_i to o_j at execution point p, whose representative is in $Closure(h_i \to h_j)$. Suppose that for some $o_k \in \mathcal{K}$ there is a path `root`... $\to o_x \to o_k$ that does not pass through o_i—that is, we have that o_i does not dominate o_k. We say that o_k is an *exposed node*. Without loss of generality we may assume that o_x is not in $\mathcal{K}$ (clearly, if we continue on the path we will reach root which is not in K). The only way o_k may flow from $\mathcal{K}$ into

o_x (or from o_x into $\mathcal{K}$) is through a "parent"—that is, there must be an edge triple such that $o_n \in \mathcal{K}$, and $o_x \rightarrow o_n$, $o_x \rightarrow o_k$ and $o_n \rightarrow o_k$. To see this, suppose that o_k flows to (or from) o_x in the other possible way, through a "sibling", and there we have on $o_k \in \mathcal{K}$ and $o_n \rightarrow o_x$, $o_n \rightarrow o_k$, and $o_x \rightarrow o_k$. This is impossible, because the representatives of these edges would have been processed at either lines 4-7 or 8-11, and thus we would have the representative of o_x in *Closure*, and thus o_x in $\mathcal{K}$. We say that edge $o_n \rightarrow o_k \in \mathcal{K}$ is an *exposed edge.*

Thus, o_n is an exposed node as well. Making the argument that node o_n can only be exposed through a "parent" leads to an exposed edge $o_{n-1} \rightarrow o_n$. Repeating the argument leads to an exposed edge $o_i \rightarrow o_1$ whose representative $h_i \rightarrow h_1$ is an outside edge. Thus, there is a contradiction because if the computation of *Closure* encounters such an edge, the algorithm returns false (lines 6 and 10).

Next, note that there is an order of exposure—that is, when o_k is exposed through node o_n we must have that o_n is exposed before o_k. Clearly, if there is a cycle of exposed edges, the cycle is ordered as well (i.e., edge $o_{n-1} \rightarrow o_n$ was exposed before $o_n \rightarrow o_k$, etc.). Therefore, some node must be exposed due to an exposed edge *not* in the cycle. Since at execution point p there are finite number of nodes and edges in $\mathcal{K}$, the exposed path will reach o_i.

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 $O(N^3)$.

The complexity of the ownership client is dominated by the ownership inference in Figure 5.3. For each edge $h_i \rightarrow h_j$ there are at most $O(N)$ objects h_k that are processed on the worklist. Furthermore, for each o_k there are at most $O(N)$ h_j objects and at most $O(N)$ h_n objects (processed at lines 4-7 and at lines 8-11). Thus, for each edge the analysis does $O(N^3)$ work. There are $O(N^2)$ edges and therefore the complexity of ownership inference is $O(N^5)$.

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 $Mod(EnclMethod(s))$

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

Figure 5.4: Immutability inference: computing the read-only status of $h_i \rightarrow h_j$.

5.4 Immutability Analysis

5.4.1 Immutability Inference

The immutability inference is presented in Figure 5.4. Lines 0-5 perform standard side-effect analysis [18, 19] 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 by modified during the invocation of m.

Finally, lines 6-9 take an edge $h_i \rightarrow h_j \in Ag$ as input and attempt to show that for all run-time edges $o_i \rightarrow o_j$ represented by this edge o_i has read-only access to o_j. The analysis examines each method call $r.m(...)$ on receiver h_i (i.e., $h_i \in Pt(r)$). $TrClosure(h_j)$ denotes the transitive closure of h_j on the points-to graph—that is, the set of all nodes reachable from h_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 h_j intersects with the set of modified objects of the run-time target of the call (i.e., $\mathit{target}(h_i, m)$), the analysis determines that edge $h_i \rightarrow h_j$ is not immutable. If this intersection is always empty, the analysis determines that $h_i \rightarrow h_j$ 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 *Hsli* points to *Hps1* and field *price* of *Hps1* points to *Hm1* (*Hm1* represents the `Money` object that holds the price of the product). Thus, `getSubtotal` calls method `times` on *Hm1*. The analysis correctly determines that *Mod*(`times`) equals {`times.this`}—that is, `times` *changes* the value of the receiver object. Thus, *Mod*(`getSubtotal`) equals {`times.this`} and we have that *Hm1* is included in set *Pt*(*Mod*(`getSubtotal`)).

Consider the call to method `getSubtotal` in `getTotal` in `Sale`, and the effect of this call on edge $\mathit{Hsli} \xrightarrow{spec} \mathit{Hps1}$. The intersection of the set of objects modified by `getSubtotal` and the transitive closure of *Hps1* is non-empty; it includes *Hm1*. 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 [1]; 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`.

5.4.2 Improved Immutability Inference

The algorithm in Figure 5.4 may incur substantial imprecision. Consider the code in Figure 5.5. Field $b1$ is immutable in A. The B object created in `main` and referred by field $b1$ is denoted by name H_{b1}, and the B object created in `m` is denoted by H_{b2}. *Mod*(`setField`) equals {`setField.this`}; it is propagated to *Mod*(`m`) and we have that *Mod*(`m`) equals {`setField.this`} as well. The points-

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

Figure 5.5: Imprecision of immutability inference.

to set of `setField.this` contains both H_{b1} and H_{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), the analysis "maps" modified formal parameters to their corresponding actuals. More precisely, it examines every variable $v \in \mathit{Mod}(m')$. 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 $\mathit{Mod}(\mathit{EnclMethod}(s))$; otherwise v itself is added to $\mathit{Mod}(\mathit{EnclMethod}(s))$.[6] Consider again the code in Figure 5.5. When propagating Mod(`setField`) to Mod(`m`) the analysis maps `setField.this` to the actual argument at the call, namely variable `b2`. As a result Mod(`m`) equals {`b2`}. Since `b2` points to H_{b2} only, the intersection of the transitive closure of H_{b1} and $\{H_{b2}\}$ is empty and the analysis concludes that $b1$ is immutable in A.

5.4.3 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 $O(N^3)$.

The complexity of the immutability client is dominated by the checking of edges (lines 6-9 in Figure 5.4). The computation of the transitive closures of all nodes is $O(N^3)$. For each edge $h_i \rightarrow h_j$ the analysis processes at most $O(N)$ calls. For each

[6] Implicit parameter `this` cannot be assigned, and other formal parameters are rarely assigned.

call it does $O(N)$ work at line 8 checking whether each $h' \in Pt(Mod(target(h_i, m)))$ is in $TrClosure(h_j)$. Since there are $O(N^2)$ edges, the complexity of immutability inference is $O(N^4)$. The entire analysis is dominated by the ownership client and therefore has complexity $O(N^5)$.

5.5 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 [20], and builds on ideas from [21].

5.5.1 Context-insensitive Information Flow Analysis

The context-insensitive information flow analysis consists of two parts: flow graph generation, and demand-driven reachability computation. During generation the analysis examines all program statements and generates a *flow graph* $\mathcal{FG}_0$. During the reachability computation the analysis starts at a source variable s and tracks the flow of s on $\mathcal{FG}_0$.

The flow graph represents direct flows. There are four kinds of nodes: variable nodes (e.g., r), field dereference nodes (e.g., $r.f$), dummy nodes representing conditional statements (e.g., s), and method nodes representing each reachable methods (e.g., m). Consequently, 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 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.

- **Conditional statement** $if(l)$**,** $case(l)$**, or** $while(l)$[7], generates implicit flow edge $l \rightsquigarrow s_l$, to associate the conditional variable l with the conditional statement, which is represented by node s_l.
  It also generates implicit flow edge $s \rightsquigarrow s_l$, where s represents the immediate enclosing conditional statement (i.e. this statement is immediately control dependent on s); or generates implicit flow edge $m \rightsquigarrow s_l$, if this statement has no control dependence within the enclosing method.
  The implicit flow edge $s \rightsquigarrow s_l$, connects the implicit flows across the nesting conditional statements; while the implicit flow edge $m \rightsquigarrow s_l$, connects the implicit flows across the method call.

- **Assignment** $l = (...operator)\ r$ generates explicit flow edge $r \rightsquigarrow l$.
  It also generates implicit flow edge $s \rightsquigarrow l$, where the assignment statement is immediately control dependent on the conditional statement s; or it generates implicit flow edge $m \rightsquigarrow l$, if the assignment statement has no control dependence within the enclosing method m.

- **Instance field write** $l.f = r$ generates an explicit flow edge $r \rightsquigarrow l.f$.
  It also generates implicit flow edge $s \rightsquigarrow l.f$, where the field write statement is immediately control dependent on the conditional statement s; or it generates implicit flow edge $m \rightsquigarrow l.f$, if the field write statement has no control dependence within the enclosing method m.
  For every variable l' such that (i) l' is aliased with l according to the points-to analysis, and (ii) there is a read of field f through variable l'(i.e., there is a field read statement $l'' = l'.f$), generates explicit flow edge $l.f \rightsquigarrow l'.f$.

- **Instance field read** $l = r.f$ generates explicit flow edge $r.f \rightsquigarrow l$.
  It also generates implicit flow edge $s \rightsquigarrow l$, where the field read statement is immediately control dependent on the conditional statement s; or it generates

[7] l is a boolean variable. Our analysis works on the intermediate representation of the three address code. Thus the conditional variable is a boolean variable l.

implicit flow edge $m \rightsquigarrow l$, if the field read statement has no control dependence within the enclosing method m.

- **Method call** i**:** $l = r_0.m(r_1, ...)$, for each $m'(this, p_1, ..., ret)$ which is a possible run-time target according to the points-to analysis:
  It generates explicit flow edges $r_0 \rightsquigarrow this$, $r_1 \rightsquigarrow p_1$, ... and $ret \rightsquigarrow l$.
  It also generates implicit flows edges $s \rightsquigarrow this$, $s \rightsquigarrow p_1$, ... $s \rightsquigarrow m'$, where the method call statement is immediately control dependent on the conditional statement s; or it generates implicit flow edge $m \rightsquigarrow this$, $m \rightsquigarrow p_1$, ... $m \rightsquigarrow m'$, if the method call statement has no control dependence within the enclosing method m.
  The implicit flow edges $s \rightsquigarrow m'$ and $m \rightsquigarrow m'$, capture the inter-procedural implicit flows.

The handling of assignments and method calls directly follows the information flow model defined in Section 4.3. The handling of field accesses uses alias information instead of points-to information; we believe that this representation is more informative when tracking indirect information flow.

Tracking shallow flow from a source variable s amounts to a straightforward reachability computation: while there are new edges and no untrusted variable has been reached, the analysis examines pairs $s \rightsquigarrow v_1 \rightsquigarrow v_2$ and adds $s \rightsquigarrow v_2$ to the flow graph if $s \rightsquigarrow v_2$ is not already there.

Tracking deep flow from s requires tracking shallow flow from multiple sources. There is a worklist of sources which is initialized to s. While there are sources on the worklist, the analysis takes a source s' and tracks shallow flow from s'. Each s' is part of the object structure rooted at s and the shallow flow computation checks for violating flow from there. For each s' of reference type, the analysis finds each valid alias v of s' and examines field read statements $s'' = v.f$; if s'' is a new source, it is added to the worklist of sources.

Note that the information flow analysis, as described above 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 3.2.2).

5.5.2 Implicit flow

Implicit flow arises from conditional variable to any assigned variable in its scope. Intraprocedural implicit flow is captured by implicit flow edges $v \rightsquigarrow s_1$, $s_1 \rightsquigarrow$ ..., $s_n \rightsquigarrow l$ (the edges reflects several nested conditional statements s_i). Interprocedural implicit flow from conditional variable v (in conditional statement s of m) to variable l in m', which is assigned after method call dispatch of m in the scope of s, is captured by implicit flow edges $v \rightsquigarrow s$, $s \rightsquigarrow m$', m'$\rightsquigarrow l$.

As interprocedural implicit flow is broken down into local implicit flow edges, the immediate scope of conditional variable could be identified intraprocedurally. The scope identification is based on intraprocedual control flow graph(CFG). 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. Each block that contains conditional statement has at least two successors.

As shown in Figure 5.6, each block computes its stack of enclosing conditional statements based on its stack of enclosing loops, and the stacks of its predecessor.

Procedure `getLoops` obtains the enclosing loop stack of a block b. Each loop is represented by its entry block, as any path go through the loop contains the representing entry block. The enclosing loop entry stack of $b's$ predecessor is scanned, to see if b has already jumped out of those represented loops. Block b is within a loop represented by block *entry*, if and only if b reaches *entry* in CFG without passing any of the loop entries enclosing that loop. Then the procedure checks if b itself is a loop entry, i.e. b reaches b without any back edge by passing any of $b's$ enclosing loop entries.

A block b is *not* in the scope of conditional statement s of block a if and only if all successor blocks a_i of a could reach b, without through any of $b's$ enclosing loop. The fact that the reaching from a_i to b is not through any of $b's$ back edge, guarantees that b is not in any branch of conditional statement s. As in procedure

`getConditions`, the enclosing conditional stack of $b's$ predecessor is scanned, to see if b is *not* in the scope of those conditional statements. Note that for dealing with special cases of *break* statements, we first check if block b is within conditional statement's enclosing loop, as shown in lines 3-4. If b is no longer in the enclosing loop of a conditional statement, then b has broken up of the loop and thus is not controlled by that conditional statement. Finally, if b has no enclosing conditional statement, as shown in flow construction, we add its enclosing method into the conditional stack.

As shown in procedure `ConditionRange`, after obtaining enclosing loops and enclosing conditionals for a block b, the stacks are passed to $b's$ successor blocks to continue computing for the successors, unless all conditionals in stack for b has been passed to successors before. This process continues until no new conditionals needs to be dealt with any block.

5.5.3 Context-sensitive Shallow Flow Analysis

The context-insensitive 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. Section 5.5.3.1 illustrates the imprecision of context-insensitive shallow flow analysis, and Sections 5.5.3.2, 5.5.3.3 and 5.5.3.4 describe in detail the context-sensitive shellow flow analysis. Section 5.5.3.5 discusses the termination, complexity and correctness properties of the analysis.

5.5.3.1 Imprecision of Context-insensitive Analysis

Example. Consider the example in Figures 4.3 and 4.4 and let us be interested in the flow of constant `LOCFLG` defined in interface `ZipConstants`. The following

edges created by the context-insensitive analysis represent flow relevant to `LOCFLG`:

`LOCFLG` $\rightsquigarrow$ `get32.off`	(due to line 6)
`get32.off` $\rightsquigarrow$ `get16.off`	(line 14)
`get16.off` $\rightsquigarrow$ `get16.i1` $\rightsquigarrow$ `get16.ret`	(lines 12 and 13)
`get16.ret` $\rightsquigarrow$ `get32.i1` $\rightsquigarrow$ `get32.ret`	(lines 14,16-17)
`get32.ret` $\rightsquigarrow$ `readLOC.i2`	(line 8)
`readLOC.i2` $\rightsquigarrow$ `getSize.this.size`	(line 9 and alising)
`getSize.this.size` $\rightsquigarrow$ `getSize.ret`	(line 19)
`getSize.ret` $\rightsquigarrow$ `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` $\rightsquigarrow$ `get32.i1` $\rightsquigarrow$ `get32.i3` $\rightsquigarrow$ `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` $\rightsquigarrow$ `get32.off` results from the call of method `get32` at line 6, and flow edge `get32.ret` $\rightsquigarrow$ `readLOC.i2` results form the return from `get32` at line 8. Clearly, this sequence represents an invalid flow path.

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

The context-sensitive analysis consists of three parts: generation of flow graph $\mathcal{FG}_0$, summarization of the effects of callee onto callers, and demand-driven reachability propagation on the summarized graph. This analysis is based on CFL-reachability [20], and builds on ideas from [21].

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 infeasible paths.

There are no annotations on the flow edges generated for assignments and instance field reads; these are treated as in Section 5.5.1. Instance field writes and method calls are annotated as follows:

- **Instance field write** $l.f = r$ generates an explicit flow edge $r \overset{*}{\rightsquigarrow} l.f$ and implicit flow edge $s \overset{*}{\rightsquigarrow} l.f$ (s is the nearest enclosing conditional statement, or the enclosing method m if no enclosing conditional statement exists.)
  For every variable l' such that (i) l' is aliased with l according to the points-to analysis, and (ii) there is a read of field f through variable l'(i.e., there is a field read statement $l'' = l'.f$), generates explicit flow edge $l.f \overset{*}{\rightsquigarrow} l'.f$.

- **Method call** i**:** $l = r_0.m(r_1, ...)$, for each $m'(\mathit{this}, p_1, ..., \mathit{ret})$ which is a possible run-time target according to the points-to analysis, generates explicit flow edges $r_0 \overset{(_i}{\rightsquigarrow} \mathit{this}$, $r_1 \overset{(_i}{\rightsquigarrow} p_1$, ... and $\mathit{ret} \overset{)_i}{\rightsquigarrow} l$, and implicit flows edges $s \overset{(_i}{\rightsquigarrow} \mathit{this}$, $s \overset{(_i}{\rightsquigarrow} p_1$, ... $s \overset{(_i}{\rightsquigarrow} m'$ (s is the nearest enclosing conditional statement, or the enclosing method m if no enclosing conditional statement exists).

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 $\mathit{ret} \overset{)_i}{\rightsquigarrow} l$ 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, $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow \mathit{ret} \overset{)_i}{\rightsquigarrow} l_1$ is a valid path, but $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow \mathit{ret} \overset{)_j}{\rightsquigarrow} l_2$ is not.

The $*$ annotations at field writes are novel; they handle flow through objects which typically transcend calling contexts. The $*$ annotations are best explained by the following example. Suppose that there is a call site `i: r.set(k)` which sets field f of the receiver to the value of k (i.e., there is statement `this.f=p;` where `p` is the formal parameter of `set`). Later there is a call `j: l=r.get()` which returns field f of the receiver (i.e., there is statement `return this.f;`). The flow edges for these statements are: $k \overset{(_i}{\rightsquigarrow} p \overset{*}{\rightsquigarrow} \mathit{get.this.f} \rightsquigarrow \mathit{get.ret} \overset{)_j}{\rightsquigarrow} l$. The wildcard "cancels" call and return annotations. In the above example $(_i$ is concatenated with the wildcard and it is "cancelled" by it resulting in transitive flow edge $k \overset{*}{\rightsquigarrow} \mathit{get.this.f}$ and later $k \overset{*}{\rightsquigarrow} \mathit{get.ret}$. Subsequently, the wildcard "cancels" $)_j$ resulting in flow edge $k \overset{*}{\rightsquigarrow} l$.

Example. Let us continue with the edges from the previous section. With

context-sensitive analysis the edges will have annotations as follows:

$\texttt{LOCFLG} \overset{(_6}{\rightsquigarrow} \texttt{get32.off}$	(due to line 6)
$\texttt{get32.off} \overset{(_{14}}{\rightsquigarrow} \texttt{get16.off}$	(line 14)
$\texttt{get16.off} \rightsquigarrow \texttt{get16.i1} \rightsquigarrow \texttt{get16.ret}$	(lines 12-13)
$\texttt{get16.ret} \overset{)_{14}}{\rightsquigarrow} \texttt{get32.i1} \rightsquigarrow \texttt{get32.ret}$	(lines 14,16-17)
$\texttt{get32.ret} \overset{)_8}{\rightsquigarrow} \texttt{readLOC.i2}$	(line 8)
$\texttt{readLOC.i2} \overset{*}{\rightsquigarrow} \texttt{getSize.this.size}$	(line 9 and alising)
$\texttt{getSize.this.size} \rightsquigarrow \texttt{getSize.ret}$	(line 19)
$\texttt{getSize.ret} \overset{)_{23}}{\rightsquigarrow} \texttt{ph_long}$	(line 23 in `main`)

5.5.3.3 Summarization

Procedure *Summarize* in Figure 5.7 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 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:

$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \rightsquigarrow v_3) = v_1 \overset{(_i}{\rightsquigarrow} v_3$$

$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{)_i}{\rightsquigarrow} v_3) = v_1 \rightsquigarrow v_3$$

$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{*}{\rightsquigarrow} v_3) = v_1 \overset{*}{\rightsquigarrow} v_3$$

$$concat(v_1 \overset{(_i}{\rightsquigarrow} v_2, v_2 \overset{*}{\rightsquigarrow} \mathit{this}.f) = v_1 \overset{*}{\rightsquigarrow} v'.f$$

$$(v' \textit{ is alias of the receiver at call site } i)$$

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 "cancelled" 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.

Example. In our running example, procedure *Summarize* produces the 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 `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.ret` and `get32.ret` $\overset{)_8}{\rightsquigarrow}$ `readLOC.i2` are not concatenated because indices 6 and 8 do not match—clearly, these edges correspond to flows due to different contexts of invocation of `get32`.

5.5.3.4 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 our example, 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* 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, 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 4.3 and 4.4. 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`.

The algorithm for *Propagate* (shown in Figure 5.7) 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 $s \overset{(_i}{\leadsto} v$ result in path edges $s \overset{Call}{\leadsto} v$. Edges of kinds $s \leadsto v$, $s \overset{*}{\leadsto} v$ and $s \overset{)_j}{\leadsto} v$ result in path edges $s \overset{nCall}{\leadsto} v$. Each path edge $s \overset{p}{\leadsto} v_1 \in \mathcal{FG}_p$ is concatenated with edges $v_1 \overset{a}{\leadsto} v_2 \in \mathcal{FG}^*$. 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 given below:

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

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 given below:

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

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

Example. For our example edges, there would be the following path edges with source `LOCFLG`:

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

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` $\overset{nCall}{\leadsto}$ `readLOC.i1`, results from *Summarize*—there is an edge `LOCFLG` $\leadsto$ `readLOC.i1` in $\mathcal{FG}^*$ which is converted into the *nCall* path edge.

5.5.3.5 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 $(_{\dots}$ edge or a $)_{\dots}$ 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 $s \mapsto \dots o_1.f_1 \mapsto \dots o_2.f_2 \mapsto \dots r$ has an appropriate analysis representative in $\mathcal{FG}_p$. The path consists of segments $s \mapsto \dots o_1.f_1$, $o_1.f_1 \mapsto \dots o_2.f_2$, etc. where all intermediate nodes between dereferences are local variables that are unique for their creating stack frame.

Consider a segment $s \mapsto \dots o_1.f_1$. This segment can have the following structure: $s \longmapsto fr_1 \overset{ret}{\mapsto} \dots fr_k \overset{call}{\mapsto} \dots fr_n \mapsto o_1.f_1$, 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 $v \overset{call}{\mapsto} fr_{k+1} \ldots fr_n \mapsto o_1.f_1$ 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 the accompanying technical report [22]. 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) $v \overset{call}{\mapsto} fr_{k+1} \ldots fr_n \mapsto o_1.f_1$ could be recursive.

Thus, segment $s \mapsto fr_1 \overset{ret}{\mapsto} \ldots fr_k \overset{call}{\mapsto} \ldots fr_n \mapsto o_1.f_1$ will have analysis representative(s) $s \overset{nCall}{\rightsquigarrow} p_1.f_1$ in $\mathcal{FG}_p$. One can see that the sequence $s \mapsto \ldots o_1.f_1 \mapsto \ldots o_2.f_2 \mapsto \ldots r$ 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.

5.5.4 Deep Flow Analysis

This section considers *deep flow* analysis. Section 5.5.5 presents the algorithm for deep flow computation for the purpose of detecting confidentiality violations. Section 5.5.6 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_` 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 sen-

sitive 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 5.7: $FShallow(s) = \{v \mid s \overset{p}{\leadsto} v \in \mathcal{FG}_p\}$. 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: $v_1 \overset{p}{\leadsto} s$ on the worklist, and examines edges e_2: $v_2 \overset{a}{\leadsto} v_1$ 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 and to the worklist. The details on this computation are given in [22].

The union of *FShallow*(s) and *BShallow*(s) gives the set of valid aliases of a reference variable s.

5.5.5 Confidentiality Inference

Recall from Section 4.3 that if a sensitive variable s is a reference variable, there may be flow from the object structure rooted at s. By definition (Section 4.3), 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 5.8 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 4.3 4.4 in Section 4.3. Source node s is `tmpbuf` and *Sinks* includes all placeholder variables from `main` in Figure 4.4. 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 4.3 and `get16.b1` is added to SWL. The algorithm proceeds to compute $FShallow(\texttt{get16.b1})$ 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 4.3 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.

5.5.6 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 5.8. It has the same inputs as the algorithm for confidentiality inference and differs only slightly (the two adjustments are highlighted in bold):

[3] if $\mathbf{BShallow(s)} \cap \mathit{Sinks} \neq \emptyset$ return false

[4] if s is of reference type

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

[6] foreach indirect **write** $\mathbf{v.f = s'}$ do

[7] add s' to SWL

Line 3 considers set $\mathit{BShallow}(s)$—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'.

procedure **ConditionRange**
input $\mathcal{CFG}_m$: control flow graph for method m
output $Range$: enclosing conditionals for each block
initialize $WL = \{\{b_{entry}, \emptyset, \emptyset\}\}$, $Stack = \{\{b_{entry}, \emptyset\}\}$
(b_{entry} is the entry block of m)
[1] while $WL \neq \emptyset$ do
[2] remove $list$: $\{b, pre_{conditions}, pre_{loops}\}$ from WL
[3] b_{loops} = getLoops(pre_{loops})
[4] $b_{conditions}$ = getConditions($pre_{conditions}$,b_{loops})
[5] if $\forall a \in b_{conditions}$, $a \in Stack(b)$ then continue
[6] $Stack(b) = Stack(b) \cup b_{conditions}$
[7] $Range(b) = Range(b) \cup \{b_{conditions}.top\}$
[8] $new_{conditions} = b_{conditions}$, $new_{loops} = b_{loops}$
[6] if b contains conditional statement s then
[7] push b to $new_{conditions}$
[8] foreach successor b' of b do
[9] add $\{b', new_{conditions}, new_{loops}\}$ to WL

procedure **getLoops**
input b: a block
pre_{loops}: stack of loop entries of predecessor
output b_{loops}: a stack of b's enclosing loop entries
initialize $b_{loops} = pre_{loops}$
[1] while $b_{loops} \neq \emptyset$ do
[2] remove b_{entry} from b_{loops}
[3] if $\exists p, b \xrightarrow{p} b_{entry}$ $(\forall a \in b_{loops}, a \notin p)$
[4] push b_{entry} in b_{loops}, break loop
[5] if $\exists p, b \xrightarrow{p} b$ $(\forall a \in b_{loops}, a \notin p)$
[6] if $b \notin b_{loops}$, push b in b_{loops}

procedure **getConditions**
input b: a block
$pre_{conditions}$: stack of conditionals of predecessor
b_{loops}: stack of b's enclosing loop entries
output $b_{conditions}$: a stack of b's enclosing conditionals
initialize $b_{conditions} = pre_{conditions}$
[1] while $b_{conditions} \neq \emptyset$ do
[2] remove b_c from $b_{conditions}$
[3] $top = loops(b_c).top$
[4] if $top \neq null \&\& top \notin b_{loops}$ then continue
[5] foreach successor b' of b_c
[6] if $\nexists p$, $b \xrightarrow{p} b_c$ $(\forall a \in b_{conditions}, a \notin p)$ then
[7] push b_c in $b_{conditions}$, return
[8] push m in $b_{conditions}$, return

Figure 5.6: Computation of conditional range.

procedure **Summarize**
input $\mathcal{FG}_0$: flow graph
output $\mathcal{FG}^*$: summarized $\mathcal{FG}_0$
initialize $\mathcal{FG}^* = \mathcal{FG}_0$
$WL = \{v_1 \overset{a}{\rightsquigarrow} v_2 \in \mathcal{FG}_0$ s.t. a is $(_i$ 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

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 5.7: Computation of shallow flow from 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 5.8: Computation of deep flow.

CHAPTER 6
Empirical Results

The static analysis framework is implemented in Java using Soot 2.2.3 [23] and Spark [16]. 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 max heap size of 800MB.

Native methods are handled by utilizing 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 [24].

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 reverse engineering of constraints on UML class diagrams: 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 tempered), while in fact f is safe. Developers could spend valuable time examining potentially large amount of code until they determine that the warning is due to analysis imprecision and not to insecure information flow. Recall that the analyses are safe—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.

6.1 Software Components

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

Table 6.1: 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 [11] and [12][8]). The components are described in the first three columns of Table 6.1. Each component contains the set of classes in *Cls* (i.e., the classes that 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.

6.1.1 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 5.3 to instance fields in functionality classes. The results are reported in Table 6.2. We applied the **immutability analyses** described in Section 5.4, 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 6.3. Finally, we applied the **information flow analyses** described in Section 5.5 on sensitive fields (i.e., non-

[8]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.

public fields) in functionality classes. The results from confidentiality and integrity inference are shown in Table 6.4.

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 6.2: Owned fields.

For each of the three analyses we examined manually the reported results. We examined each non-owned field, mutable field/parameter/method, and leaked/tempered 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 tempered are indeed leaked or tempered.

6.1.2 Analysis Cost

In this section we report on the cost of the analyses. The cost of the three analyses is given in Table 6.5. 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	#Fields (reference)	#Immutable Fields	#Methods	#Immutable Methods	#Parameters	#Immutable Parameters
`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 6.3: Immutable fields, methods and parameters.

Program	#Fields (non-public)	#Leaked (shallow)	#Leaked (shallow or deep)	#Tempered (shallow)	#Tempered (shallow or deep)
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 6.4: Confidentiality (fields leaked to client code) and integrity (fields tempered by client code).

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 6.5: 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 [25]. 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.

6.2 Complete Programs

Our benchmark suite includes several relatively small applications, `soot-c` and `sablecc-j` from the Ashes suite [26], relatively large benchmarks from the DaCapo benchmark suite version beta051009 [27] and the Polyglot Java front-end. The suite is described in Table 6.6. The number of user classes and user methods fetched by 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

(1)Program	(2)Description	(3)Size		
		User Classes	User Method	#Reachable Methods
jdepend-2.9.1	A quality metrics suite for Java	17	225	3962
javad	Classfile decompiler	41	156	3838
JATLite-0.4	Template for writing software agents	45	442	6279
undo	Undo functionality for sysadmins	237	1709	5644
hsqldb-1.8.0	Relational database engine and tools	196	3743	7177
soot-c	Analysis framework for Java	579	2935	6046
sablecc-j	Java parser generator	300	2024	7970
polyglot-1.3.2	Framework for Java language extensions	267	3418	7449
antlr	Parser and lexical analyzer generator	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 6.6: Information about the Java benchmarks.

last column shows the number of methods (user and library), determined to be reachable by Spark.

We applied the **ownership analysis** described in Section 5.3 to non-static instance fields in non-library classes[9]. The results are reported in Table 6.7. We applied the **immutability analyses** described in Section 5.4, on non-static instance fields in non-library classes, public methods in public classes, and parameters of those methods[10]. The results are reported in Table 6.8. Finally, we applied the **information flow analyses** described in Section 5.5 on sensitive fields (i.e., non-public non-static fields) in functionality classes. The results from confidentiality and integrity inference are shown in Table 6.9.

On average, the ownership analysis identified 28% of the fields as owned (column #Owned). Also, on average, the immutability analysis identified 27% of the fields as read-only (column #Immutable). And 43% of the examined data were determined to be `deep leaked`, 24% were determined to be `deep tampered`.

[9] Our experiments exclude fields of type `String` because they do not correspond to associations in the UML class diagram.

[10] Our experiments exclude fields of type `String` because they do not correspond to associations in the UML class diagram.

Program	#Instance Fields (reference type)	#Owned Fields
`jdepend`	33	19 (58%)
`javad`	40	19 (48%)
`JATLite`	142	35 (27%)
`undo`	325	73 (22%)
`hsqldb`	383	89 (23%)
`soot`	340	77 (23%)
`sablecc`	304	30 (10%)
`polyglot`	435	51 (12%)
`antlr`	161	45 (28%)
`bloat`	529	81 (15%)
`jython`	215	69 (32%)
`pmd`	914	318 (35%)
`ps`	19	7 (37%)
`Average`		24%

Table 6.7: Owned fields.

Program	Fields (reference type)	Immutable Fields	Methods	Immutable Methods	Parameters	Immutable Parameters
`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%)
`hsqldb`	383	70 (18.3%)	155	29 (18.7%)	347	83 (23.9%)
`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 6.8: Immutable fields, methods and parameters.

(1)Program	(2)#Fields	(3)#Leaked (shallow)	(4)#Leaked (shallow/deep)	(5)#Tampered (shallow)	(6)#Tampered (shallow/deep)
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%)
hsqldb	917	183(20%)	383(42%)	154(17%)	322(35%)
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 6.9: Columns 3 and 4: Confidentiality; Columns 5 and 6: Integrity.

6.2.1 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 [10, 12] 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 four largest benchmarks, `hsqldb, polyglot, sablecc` and `pmd` (the size metric that we used was the number of reachable methods, shown in Column (3) of Figure 6.6). We examined each non-owned field, mutable field/parameter/method, and leaked/tempered field, and attempted to find an execution path that would

expose appropriate non-ownership, mutability or information flow. Of 88 reported as non-owned field, we were able to show exposure that is, for this set of fields the ownership analysis achieved perfect precision. Similarly, of 97 non-readonly field, in all but 7 cases we were able to show mutability. 85 fields were reported as leaked (column (4) in Table 6.9). We examined manually 85 fields that are reported as leaked, and in *all but 2 cases* we were able to show the field leakage. Furthermore, we examined 112 fields that are reported as being tampered, and in *all but 4* case we successfully found execution paths that tampers the field. Thus the analysis is achieved very good precision as well.

6.2.2 Analysis Cost

In this section we report on the cost of the analyses. The cost of the three analyses is given in Table 6.10. The ownership analysis typically runs within 3 minutes. 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 within 15 minutes, due to the large number of parameters (over 2000). Finally, the information flow analysis, which includes both confidentiality and integrity inference, runs in about 8 minutes.

Program	Points-to Analysis	Ownership Analysis	Immutability Analysis	Information Flow Analysis
`jdepend`	1m35s	32s	20s	12s
`javad`	1m33s	27s	4s	9s
`JATLite`	2m37s	1m29s	27s	59s
`undo`	3m3s	1m52s	1m13s	46s
`hsqldb`	2m57s	2m15s	2m48s	3m47s
`soot`	2m23s	1m13s	1m59s	1m31s
`sablecc`	3m5s	1m49s	4m30s	1m39s
`polyglot`	9m39s	2m44s	9m22s	7s
`antlr`	2m25s	1m4s	68s	1m12s
`bloat`	2m36s	1m57s	14m21s	7m38s
`jython`	1m58s	1m21s	4m21s	3m48s
`pmd`	4m17s	2m22s	3m41s	47s
`ps`	2m19s	1m51s	29s	27s

Table 6.10: Analysis time.

6.3 Conclusions

The empirical study leads to the following observations. First, the analyses scale to large programs, analyzing close to ten thousand reachable methods in only several 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 led us to a bug in the code for our motivating example. Overall, the analyses produce useful results, easy to interpret in the context of UML class diagrams. Third, the analyses are relatively precise, rarely missing `owned` and `read-only` associations. In summary, the empirical study indicates that the analyses can effectively support model-driven development and reasoning about software quality and security.

CHAPTER 7
Related Work

The ownership and immutability inference analyses improve substantially upon our previous work on composition inference [12] and side-effect analysis [19] respectively. 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 are 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., [6, 2, 7]), immutability type systems (e.g., [8]) and type systems for secure information flow (e.g., [28, 9]). 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 [25, 29, 30, 31] as well as the work by Gueheneuc and Albin-Amiot [32] on dynamic analysis for inference of aggregations and compositions for reverse-engineered UML class diagrams. Work on inference of immutable parameters and methods includes [33, 11, 34], and work on inference of information flow includes [13].

7.1 Type systems

Our work is related to work on ownership type systems [6, 2, 7, 5, 35, 36] and work on immutability type systems [37, 38, 8]. 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 [39, 40, 28, 41, 42]. 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. [43] present an analysis for Java that infers whether a class is confined within its package. Clarke et al. [44] present a confinement checking tool, related to [43], 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 pseudo-generic containers well), and do not address the kind of ownership needed for UML-based object access control.

Heine and Lam [45] present an ownership inference algorithm for the purposes of memory leak detection. Their notion of ownership is substantially different than the notion of owners-as-dominators used in our work.

Aldrich et al. [7] 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 [7] (e.g., the `owned` type in [7] captures exclusive ownership only, although access can be allowed through user-specified alias parameters). The inference analysis in [7] 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 [46] infer ownership types for race-free Java using dynamic analysis; thus, the inferred types may be unsound. In contrast, our inference analysis is a safe static analysis.

Recent work by Rayside et al. [47] 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. [48] 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, analyses large libraries and focuses on instance fields.

Ryder et al. [18] 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 [11], and Salcianu and Rinard [33] 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 applied on substantially larger programs (close to ten

thousand reachable methods).

7.4 Static information flow analysis

Genaim and Spoto [13] 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 conceptually different: it is cubic, and based on CFL-reachability which we conjecture, achieves the right 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. [49, 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 conceptually different: the analysis in [49, 50] is exponential (due to the underlying points-to analysis), while ours is cubic.

Related type inference techniques have been proposed [51, 52, 53]. 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 [54, 55, 56], tools that detect SQL-injection attacks [57, 58, 59], and tools that enforce data confidentiality [60, 61, 62, 63]. Recently, Clause et al. have proposed a general framework for dynamic tainting [64]. Dynamic tainting is a principally different approach to secure information 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 [20]; it has been used in a variety of flow analyses that require context sensitivity (e.g., points-to analysis for Java [24], and analysis for race detection for C [65]). 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 right degree of scalability and precision for the problem of static information flow analysis.

Our work builds on the ideas in [21]. Unlike [21], 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 [66], 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 the $*$-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 universally-known UML class diagrams, and UML-based reasoning about important security-related properties. Furthermore, 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 ([29, 13]).

CHAPTER 8
Conclusions and Future Work

This paper advocates a practical framework for specification and verification of security-related program properties. Specifically, we propose the use of ownership, immutability, and information flow constraints on UML class diagrams, and verification of these constraints through static analysis incorporated in a reverse engineering tool.

In the future, we plan to extend our framework with other interesting constraints and corresponding analyses. Also, we plan to incorporate the analysis into an Eclipse-based reverse engineering tool for UML class diagrams.

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

[15] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

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

[17] Y. Liu and A. Milanova. UML-based alias control. Technical Report RPI/DCS-06-10, Rensselaer Polytechnic Institute, September 2006.

[18] 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.

[19] 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.

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

[21] Jacob Rehof and Manuel Fahndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[22] Y. Liu and A. Milanova. Static information flow analysis for Java. Technical Report 08-03, Rensselaer Polytechnic Institute, February 2008.

[23] 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*, pages 18–34, 2000.

[24] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

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

[26] Ashes suite collection. http://www.sable.mcgill.ca/software.

[27] Dacapo benchmark suite. http://www-ali.cs.umass.edu/ dacapo/gcbm.html.

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

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

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

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

[32] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *OOPSLA*, pages 301–314, 2004.

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

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

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

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

[47] 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.

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

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

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

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

[61] 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.

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

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

[64] 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.

[65] 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.

[66] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

APPENDIX A
Example Code

```
    public class Register {
      private ProductCatalog catalog;
      private Sale sale;
      public Register() {
1         catalog = new ProductCatalog();                 // h_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();                              // h_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();          // h_Hashtable
      ProductCatalog() {
9         ItemID id = new ItemID(100);                    // h_ItemID1
10        Money price = new Money(3);                     // h_Money1
```

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

  class Sale {
14  private Vector lineItems = new Vector();         // h_Vector
    private Payment payment;
    public Money getBalance() {
15      return payment.getAmount().minus(getTotal());
    }
    public void makeLineItem(ProductSpec s, int q) {
16      lineItems.add(new SalesLineItem(s,q));       // h_SaleLineItem
    }
    public Money getTotal() {
17      Money total = new Money();                   // h_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);                 // h_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 Mondy getPrice() { return price; }
29    public String getDescription() { return description; }
    }

    public class phMain() {
      public static void main() {
30        int q = 0, amount = 0;
```

```
31        ItemID id = new ItemID(q);                 // h_ItemID2
32        Money cash = new Money(amount);            // h_Money3
33        Register register = new Register();        // h_Register
34        register.makeNewSale();
35        register.enterItem(id,q);
36        register.makePayment(amount);
37        register.endSale();
      }
    }
```