Practical Static Analysis for Inference of Security-Related Program Properties

Yin Liu
Department of Computer Science
Rensselaer Polytechnic Institute
liuy@cs.rpi.edu

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Abstract

We present a static analysis framework for inference of security-related program properties. Within this framework we design and implement ownership, immutability and information flow inference analyses for Java.

We perform empirical investigation on a set of Java components, and on a set of established security benchmarks. The results indicate that the analyses are practical and precise, and therefore can be integrated in program comprehension tools that support reasoning about software security and software quality.

1 Introduction

Unintended object access and unintended information flow can seriously compromise software quality and software security. For example, in Java 1.1 the security function `Class.getSigners` mistakenly returned a reference to an internal array; untrusted clients could modify this array and compromise the security of the system. The problem is especially relevant in web applications, which often exhibit vulnerabilities due to unintended information flow [14]. Current languages such as Java do not provide mechanisms for preventing unintended object access and unintended information flow; therefore it is important to study reasoning techniques that can help alleviate this problem.

Our paper proposes a static analysis framework for inference of security-related properties in Java programs. Specifically, we propose *ownership*, *immutability* and *information flow* inference analyses; these analyses reveal information about object access and information flow in the program, and may help uncover serious vulnerabilities.

The *ownership property* is inferred on instance fields and variables of class type. A field/variable of type B in class A is *owned*, if every A object controls the B object it references through this field/variable (i.e., the A object may create a B object, pass the B object to other parts of its representation, but cannot expose the B object outside).

The *immutability property* is inferred on method parameters, methods, instance fields and variables of class type. A parameter p in method m is inferred as `read-only` if no invocation of m modifies the heap structure rooted at the object referred by p. A method m is inferred as `read-only` if no invocation of m modifies the visible state. A field/variable of type B in class A is inferred as `read-only` if no A object ever modifies the heap structure rooted at the B object referenced through this field/variable.

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

The proposed framework and inference analyses work directly on Java programs and do not require annotations by the programmer. They work on complete programs and on incomplete programs (i.e., software components). This is an important feature because the problem of analysis of incomplete programs arises often (for example, given a set of interacting classes such as a secure server-side component, we are interested if there could be compromising object access or information flow triggered by a client).

The proposed framework and inference analyses have several applications. They can be used to (i) verify specifications, (ii) visualize ownership, immutability and information flow (e.g., as part of reverse-engineered UML diagrams), and (iii) infer ownership, immutability or information flow types for the purposes of type checking. The analyses can be used during software development and maintenance to reveal important information about object access and information flow; they may uncover vulnerabilities.

We have implemented the framework and inference analyses. Our empirical results demonstrate that the analyses are practical and precise and uncover vulnerabilities in real-world web applications. Therefore, they can be incorporated in program comprehension tools that support reasoning about software security and software quality.

The contributions of this work are the following:

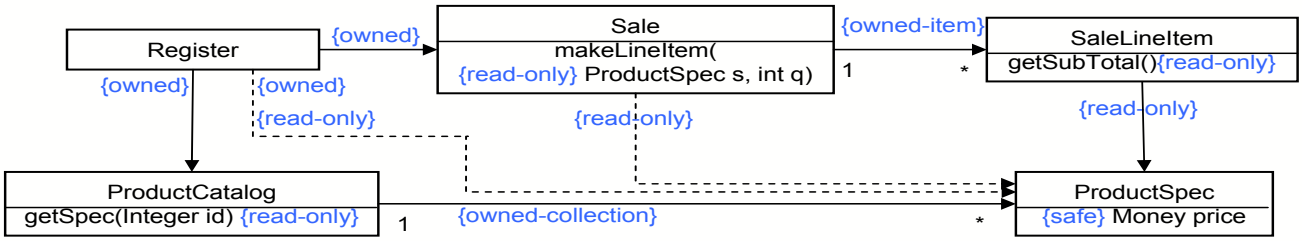

Figure 1. UML class diagram with ownership, immutability and information flow annotations.

- We present a static analysis framework for inference of security-related properties. Within this framework, we present run-time models and analyses for several security-related properties: ownership, immutability and information flow.

- We present an empirical study on a set of Java components and on a set of Java web applications established as security benchmarks [14, 1]. The study shows that the analyses are practical and precise, and therefore, can be integrated in program comprehensive tools.

2 Motivating Example

As a motivating example, consider the UML class diagram in Figure 1. It illustrates the design of a component of a Point-of-Sale (POS) system [8]. The solid lines represent permanent associations (implemented through instance fields), and the dashed lines represent temporary dependencies (typically implemented through local variables). The Java classes for this example, taken from [8] with minor modifications, are given in Appendix A.

We have added ownership, immutability and information flow annotations to the UML class diagram. These annotations formalize the design and security requirements for the POS system. They represent the desired properties of the implementation of this design.

A `Register` object, an 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.

Let us consider several of the specified annotations. The `owned` annotations between `Register` and `Sale` specifies that the `Sale` object is part of the internal representation of the `Register` object and should not be leaked outside (i.e., to potentially untrusted clients). The `read-only` annotation on the association between `SaleLineItem` and `ProductSpec` specifies that `SaleLineItem` has read-only access to `ProductSpec` (i.e., it forbids `SaleLineItem` from modifying `ProductSpec`). The annotation on parameter `s` (of type `ProductSpec`) in method `makeLineItem` in class `Sale`, specifies that `makeLineItem` has read-only access to `s` (i.e., it forbids `makeLineItem` from modifying the `ProductSpec` object referred by `s`). The last two annotations formalize the design decision that the `ProductCatalog` is the "information expert" and the only object that can initialize and update product information. The `read-only` annotations on methods `getSpec` in class `ProductCatalog` and `getSubtotal` in class `SaleLinItem` specify that these methods do not modify certain visible state.

The `safe` annotation on field *price* (of type `Money`) in `ProductSpec` 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). Note that `ProductSpec` objects are owned by the `Register` object; however, ownership does not prevent deeper information flow violations such as leaks or modifications to sensitive data that is part of the owned `ProductSpec` objects.

Our static analysis infers these properties. Recall the code in Appendix A. Our analysis infers that field `sale` in class `Register` is `owned`; this property is visualized as `owned` annotation on the association between `Register` and `Sale` in the reverse-engineered UML class diagram. Visualizing ownership, immutability and information flow as part of reverse-engineered UML class diagrams, could help verify security-related design requirements and lead to higher quality, more secure and understandable software.

3 Run-time Models

We consider ownership, immutability and information flow properties as outlined in the previous sections. There are two issues. First, we need to define run-time models for these properties (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 property c, does there exist an execution of the program 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?).

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

3.1 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [5, 4, 19]. 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 [5].

In this model, program execution 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' if and only if at some point of program execution one of the following is true: (1) field f of o refers to o', or (2) a method m invoked with receiver o has a local variable r which refers to o'. We say that *o owns o'* if and only if o is the immediate dominator of o' in the object graph. Consequently, a field f or variable v in class A of class type B is `owned`, if and only if for every program execution, each instance of A owns the corresponding instances of B that it references through f or v.

3.2 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'').

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. 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.[1]

Finally, we say that o has read-only access to o' if no execution of a method on receiver o modifies o'. A field f or variable v of type B in class A is `read-only`, if and only if for every program execution, each instance of A has read-only access to the corresponding instances of B.

3.3 Information Flow Model

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. Such flows are *direct* and *indirect* [6, 7]. Direct flows can be *explicit* (i.e., data-flow based) and *implicit* (i.e., control-flow based). Indirect flows arise from compositions of direct flows. We consider *only explicit flow* which we conjecture, is suitable for the purposes of program comprehension.

We consider the following information flows due to Java statements. An assignment statement $l = (...operator)\ r$ leads to flow $r \mapsto l$. An instance field write statement $l.f = r$ leads to flow $r \mapsto o.f$ (o is the run-time object referred by l at the point of execution of the statement). An instance field read statement $l = r.f$ leads to flow $o.f \mapsto l$ (again, o is the run-time object referred by r at the point of execution of the statement). Finally, a method call statment $l = r_0.m(r_1, ...)$ dispatched to method $m'(this, p_1, ..., ret)$ leads to flows $r_0 \mapsto this$, $r_1 \mapsto p_1$, ... and $ret \mapsto l$ ($this$ denotes the implicit parameter `this` of m', p_i denote the formal parameters of m', and ret denotes a special variable that holds the return value of m').

There are two types of indirect flow: shallow flow and deep flow. There is *shallow flow* from variable l to 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 $l_1.f = l$ leads to flow $l \mapsto o.f$, and then the execution of $l_3 = l_2.f$ (l_1 and l_2 both point to object o) leads to flow $o.f \mapsto l_3$. Finally the execution of $r = l_3 - y$ leads to flow $l_3 \mapsto r$.

Note that when l is a reference variable, there may be flow from the object structure rooted at l. There is *deep flow from the structure of l to r* if and only if there is shallow flow from some l' to r, where l' is an alias of some $l.f_1.f_2...f_k$. Similarly, when r is a reference variable, there may be flow into the object structure rooted at r. There is *deep flow from l into the structure of r* if and only if there is shallow flow from l into some r', where r' is an alias of some $r.f_1.f_2...f_k$.

Certain program variables are designated as untrusted *sinks* and other variables are designated as untrusted *sources*; the selection of sinks and sources depends on the security problem. We say that a field f or variable v in class A is `confidential`, if and only if there is no program

[1] 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. The choice is arbitrary—the definition and corresponding analyses can be trivially changed to accommodate other choices.

execution for which there is deep flow from the structure of rooted at f or v to a sink. Similarly, a field f in class A is `safe` if there is no information flow from a source to the structure rooted at f or v. Section 4.3.3 and Section 5 give concrete examples of potentially harmful information flow.

4 Static Analysis Framework

Our static analysis framework handles (i) analysis of complete programs (whole programs), and (ii) analysis of incomplete programs (software components). In the latter case, the component is defined as a set of interacting classes Cls with designated *accessible* methods and fields; a client accesses *Cls* through these accessible methods and fields.

Our framework is based on whole-program analysis. We address the analysis of incomplete programs by employing a general technique called *fragment analysis* [24]. The fragment analysis reduces the problem from analysis of an incomplete program to an analysis of a complete program. The fragment analysis is described in Section 4.1. Furthermore, the analysis problems require points-to information and we employ a general-purpose points-to analysis. The points-to analysis is described in Section 4.2.

The fragment analysis and the points-to analysis are a general foundation that allows building of different client analyses. These analyses form an inexpensive foundation — they are dominated by Andersen's analysis which has complexity $O(n^3)$. So far we have built analyses that infer ownership, immutability and information flow in accordance with the models outlined in the previous section. These analyses are relatively inexpensive, both in terms of worst-case complexity and in terms of running times. The analyses are described in Section 4.3. We envision that the framework will be augmented with other properties of interest and corresponding client analyses.

4.1 Fragment Analysis

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 [24]. The `main` method for our running example is given at the end of Appendix A.

4.2 Points-to Analysis

Points-to analysis is a well-known program analysis. It finds the objects that a given reference variable or a reference object field may point to. Points-to information is needed by all of our client analyses; most likely it will be needed by future client analyses as well. There is a wide variety of points-to analyses, with different degrees of precision and cost. Our work uses Andersen's points-to analysis [23, 9]. 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). Andersen's analysis is cubic, and it scales to large programs.

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" a 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 [24].

4.3 Client Analyses

So far we have developed ownership, immutability and information flow inference analyses for Java within this general framework. 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 properties in accordance with the models presented in Section 3.

Next we briefly describe our analyses with illustrating examples. 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.

4.3.1 Ownership Client

Using the points-to graph, the ownership analysis first constructs the *approximate object graph* Ag, which approximates all possible run-time object graphs. The nodes in Ag are object names, and the edges represent "may-access" relationships. Subsequently, the ownership analysis uses Ag to reason about ownership. It examines an edge $h_i \rightarrow h_j$ in Ag, and finds all the paths from h_i to h_j. If all these paths are confined within the ownership boundary of h_i, the analysis concludes that for *each* run-time edge $o_i \rightarrow o_j$ represented by $h_i \rightarrow h_j$, o_i dominates and therefore owns o_j.

Recall the code in Appendix A. In the Ag for this code, there is an edge from $h_{Register}$ (the object name that corresponds to line 33 and represents the instances of class `Register`) to h_{Sale} (the object name that corresponds to line 4 and represents the instances of class `Sale`); in this case $h_{Register}$ is the only object that could access h_{Sale}, and the analysis concludes that $h_{Register}$ owns h_{Sale}. The analysis infers that field `sale` in class `Register` is `owned` and the UML association between `Register` and `Sale` is reverse engineered as `owned`. Thus, the implementation

meets the requirement stated in Figure 1. As another example, consider the edge from $h_{Register}$ to $h_{ProductSpec}$; there are many access paths to $h_{ProductSpec}$ since $h_{ProductSpec}$ is passed to h_{Sale} and further down. However, all these access paths are internal to the boundary of $h_{Register}$ and the analysis infers that $h_{Register}$ owns $h_{ProductSpec}$. Thus, the UML association between `Register` and `ProductSpec` is `owned` and the implementation meets the requirement stated in Figure 1.

4.3.2 Immutability Client

The immutability analysis is based on standard side-effect analysis [25, 17]. This analysis computes a set $Mod(m)$ for each method m—this set contains the objects that may by written during an invocation of m. In addition, we compute $TrClosure(v)$—this set contains the objects transitively reachable from v on a path of field edges.

Recall that by definition a parameter p_i in method m is `read-only` if no execution of m writes an object transitively reachable from p_i. Thus, if the intersection of $Mod(m)$ and $TrClosure(p_i)$ is empty, the analysis infers that p_i is `read-only`. Consider parameter `s` in method `makeLineItem`. The analysis computes $Mod($`makeLineItem`$) = \{h_{Vector}, h_{data}, h_{SaleLineItem}\}$ (i.e., `makeLineItem` may write (1) the vector h_{Vector}, (2) the internal array of the vector h_{data}, and (3) the newly created line item object, $h_{SaleLineItem}$). According to the points-to analysis, we have $TrClosure($`s`$) = \{h_{ProductSpec}, h_{Money1}\}$ (i.e., the specification object $h_{ProductSpec}$ itself, and the money object h_{Money1} referred by its field $price$). The two sets have empty intersection, and the analysis infers that `s` is a `read-only` parameter; the implementation meets the requirement in Figure 1.

Recall that method m is `read-only` if the following two conditions are true: 1) all parameters p_i of m are `read-only`, and 2) no invocation of m modifies a static field. We have $Mod($`getSpec`$) = \emptyset$ and the analysis infers that method `getSpec` is `read-only`; again, the implementation meets the requirement in Figure 1. However, assuming a mutable `Money` class, we have that $Mod($`getSubtotal`$) = \{h_{Money1}\}$ which intersects with $TrClosure($`getSubtotal`$.this)$ and the analysis infers that `getSubtotal` is not `read-only`;[2] thus the implementation *violates* the `read-only` requirement stated in Figure 1 (the `read-only` annotation is omitted from the reverse engineered class diagram).

Finally, recall that a field f is `read-only` if every runtime instance of f's enclosing class has read-only access to its f field. Let h be the analysis name of an instance of the enclosing class of f. Our analysis considers each method m called on receiver h. If for each pair h and m we have that the intersection of $Mod(m)$ and $TrClosure(h.f)$ is empty, the analysis determines that f is `read-only`. Consider the call to `getSubtotal` at line 21. It modifies the *price* field of `ProductSpec` and the analysis determines that a `SaleLineItem` object can modify a `ProductSpec` object, which is a *violation* of the `read-only` requirement on the UML association between `SaleLineItem` and `ProductSpec` in Figure 1.

4.3.3 Information Flow Client

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

There is shallow flow from variable s to variable r, if r could be reached from s through a valid flow path in the summary flow graph. Tracking of deep flows from variable s amounts to tracking shallow flows from multiple sources; these sources are part of the object structure rooted at s.

Consider the code in Appendix A. For the purposes of this security application, variables in `main` (i.e., the client) are designated as untrusted sources and sinks. Consdier field $price$ in class `ProductSpec`. One can easily see that there is no shallow flow from `main` into this field. The only shallow flow, from local variable `price` to $h_{ProductSpec}.price$ is due to the code at lines 10 and 11 in the constructor of `ProductCatalog`. However, there could be deep flow from `main` into field $price$. First, the analysis infers shallow flow from variable `q` in `main` to $h_{SaleLineItem}.quantity$, denoted by `q` $\rightsquigarrow$ $h_{SaleLineItem}.quantity$. This flow is due to intermediate flows `q` $\rightsquigarrow$ `enterItem.q` $\rightsquigarrow$ `makeLineItem.q` $\rightsquigarrow$ `SaleLineItem.q` $\rightsquigarrow$ $h_{SaleLineItem}.quantity$. Second, the analysis infers flow $h_{SaleLineItem}.quantity \rightsquigarrow h_{Money1}.amt$ after analysis of line 25 and the code in method `times` (here amt stands for the field of simple type `double` which holds the numeric value of the money object). Consequently, the analysis infers that there is deep flow from variable `q` in `main` (i.e., in client code) to the object structure reachable from $price$ (specifically, $price.amt$) field $price$ is not `safe`. Thus, the implementation *violated* the `safe` requirement in Figure 1. The consequences of this violation, and the previous violations, could be significant—subsequent sales could fetch wrong product prices and compute incorrect sale totals.

5 Empirical Results

The static analysis framework is implemented in Java using Soot 2.2.3 [29] and Spark [9]. It uses the Andersen-

[2] Note that `Money` could be mutable or immutable. We assume a mutable implementation because `getTotal` in the textbook code wouldn't have worked with the immutable one.

(1)Component	(2)Functionality	(3)#Class in *Cls*/ #Functionality	(4)#Fields in Functionality	(5)#Rechable Methods
`gzip`	GZIP IO streams	199/6	23	3481
`zip`	ZIP IO streams	194/6	43	3506
`checked`	IO streams&checksums	189/4	3	3428
`collator`	text collation	203/15	169	3535
`breaks`	text break	193/13	252	3487
`number`	number formatting	198/10	76	3541

Table 1. Information on Java components.

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 600MB.

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 [22] and [16][3]). The results on the components are presented in Section 5.1. We also evaluated the framework and the analyses on a set of web applications established as security benchmarks [1, 14]. The results on the web applications are presented in Section 5.2.

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 information flow analysis reports safe data as tempered?). Precision is crucial: imprecise analysis is not merely useless, but also confusing, and may discourage developers from using analysis-based tools. For example, tracking information flow in web applications requires that a large amount of code in Apache is examined. 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. It is important to note 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*—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 use the tool.

5.1 Software Components

The components are described in the first three columns of Table 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. The analysis attaches a placeholder `main` to *Cls* and performs ownership, immutability and information flow analysis. Recall from Section 4.1 that `main` approximates all possible clients that can be written on top of *Cls* — therefore, the analysis results approximate over all possible clients (e.g., if a field f is inferred as `owned`, then it is guaranteed that one cannot write a client which exposes the object stored in f outside of its enclosing object).

We applied the **ownership analysis** described in Section 4.3.1 on instance fields in functionality classes. The results are reported in Table 4. We applied the **immutability analyses** described in Section 4.3.2 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 2. We applied the **information flow analyses** described in Section 4.3.3 on sensitive fields (i.e., non-public fields) in functionality classes. In our security model, the functionality classes are trusted, and the client code is untrusted. Thus, the set of sinks and the set of sources consist of all variables in placeholder `main`. If a field f in functionality class C is inferred as `confidential` (i.e., there is no deep flow from f to to a variable v in `main`), then there is no client that causes deep flow from f to the client. The results from confidentiality and integrity inference are shown in Table 3.

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`	2	1 (33%)

Table 4. Owned fields.

For each of the three analyses we examined manually the

[3]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.

Program	#Fields (reference type)	#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 2. 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 3. Confidentiality (fields leaked to client code) and integrity (fields tempered by client code).

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.

In terms of cost, all analyses scale well. The ownership analysis typically runs within 20 seconds (times range from 19s to 29s). The immutability analysis, which includes the analysis of fields, methods and parameters, runs within seconds as well (times range from 18s to 40s). The information flow analysis, which includes both confidentiality and integrity inference, runs within 11 seconds on all components.

5.2 Web Applications

We use SecuriBench [1, 14], a set of Java Web applications established as benchmarks for research on program security.[4] Information about these benchmarks is presented in Table 5.

We use the security model in [14]. In this model, the untrusted sources are return variables from particular methods (e.g., method `getParameter()` in class `javax.servlet.ServletRequest`) and parameters of particular methods (e.g., the parameter of `main`). The sources are classified in the following categories of security attacks: (i) HTTP header manipulation, (ii) parameter manipulation, (iii) cookie poisoning and (iv) non-web sources (i.e., the parameters of `main`). We specified as sources the returned variables and parameters as described in [14, 13]. The number of sources is shown in column (3) of Table 5.

The set of sensitive variables (i.e., trusted program data that should be `safe` and never tempered by flow from untrusted sources) includes arguments passed to security-sensitive methods such as SQL queries (e.g., `executeQuery(String)`), HTTP response (e.g., `sendRedirect(String)`), server-side output streams (e.g., `JspWriter.print(String)`), file paths (e.g. `File(String)`), and commands executed by the system (e.g., `Runtime.exec(String)`). The sensitive variables represent different categories of security vulnerabilities: (i) SQL injection, (ii) HTTP response splitting, (iii) cross-site scripting, (iv) path traversal, and (v) command injection(stealth commanding). Again, we specified the sensitive variables as in [14, 13]. The number of sensitive variables is shown in column (4) of Table 5; we report a larger number of sources and sensitive variables than [14] because we include non-`String` parameters, and because our analysis includes a larger number of reachable methods.

Column (5) in Table 5 shows the number of methods, including library methods, reachable by Spark.

We applied the **information flow analysis** described in Section 4.3.3, to track information flow from sources to sensitive data. The results of our analysis are shown in Table 6. Consider the cell for Parameter manipulation and SQL injection. It has entry `webgoat`: 6. This means that there are 6 pairs (p, s), where p is a source classified in the category "Parameter manipulation", and s is a sensitive variable that causes an "SQL injection" vulnerability (e.g., through a call `executeQuery(s)`), and there is deep information flow from p to s; in other words, s is not `safe`.

We manually examined the security violations reported

[4] We include 6 of the 9 benchmarks. Applications `blojsom` and `road2hibernate` were not available for download, and `snipsnap` did not run through Soot.

(1)Benchmark	(2)Version	(3)#Sources	(4)#Sensitive variables	(5)#Rechable Methods
`jboard`	0.30	1	16	4220
`blueblog`	1.0	11	39	4836
`webgoat`	0.9	10	81	5698
`personalblog`	1.2.6	31	32	9570
`pebble`	1.6-beta1	124	78	7622
`roller`	0.9.9	40	94	13623

Table 5. Information on Web application security benchmarks.

	SQL injection	HTTP splitting	Cross-site scripting	Path traversal	Command injection	Total
HTTP header manipulation	0	0	`blueblog`: 1, `webgoat`: 1, `pebble`: 1, `roller`: 1	0	0	4
Parameter manipulation	`webgoat`: 6	0	0	0	`webgoat`: 1	7
Cookie poisoning	`webgoat`: 1	0	0	0	0	1
Non-Web inputs	0	0	0	0	0	0
Total	7	0	4	0	1	12

Table 6. Classification of security violations discovered by information flow analysis.

by our analysis. In all cases, we were able to confirm the information flow from the source to the sensitive variable as reported by the analysis. *Our results are the same as the results reported in [14]*, except for 2 cases. First, our analysis discovers 3 new violations in `webgoat`; the manual examination confirmed these violations. Second, our analysis does not discover the 2 violations on `personalblog`. Overall, the precision experiments confirm that our inexpensive analysis achieves very good precision.

The cost of our information flow analysis is practical. It runs within 45 seconds on all benchmarks, except for `roller` (with about 14K reachable methods), on which it runs in 505 seconds. Our analysis appears to run faster than the analysis in [14] for most of the benchmarks.

6 Related Work

There are many proposals for language-based reasoning about ownership, immutability and information flow—there are proposals for ownership type systems (e.g., [18, 5, 2]), immutability type systems (e.g., [28]) and type systems for secure information flow (e.g., [26]). 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.

Automatic inference of ownership, immutability and information flow has received significantly less attention. Recent work on static inference of ownership-like properties includes [15], work on inference of immutable parameters and methods includes [27, 22, 3], and work on inference of information flow includes [14, 7]. The main advantage of our framework compared to previous static analyses [14, 15], is its scalability. Livshits and Lam present an information flow analysis which relies on an exponential context-sensitive points-to analysis [14]. In contrast, our information flow analysis relies on the cubic Andersen's analysis and has cubic worst-case complexity [11]; in the same time, our analysis achieves comparable precision to [14].

The novelty of our work is that it presents an extensible, scalable, static analysis framework which allows automatic inference of different kinds of properties. It generalizes our previous work on ownership, immutability and information flow analysis [10, 12]. Additionally, this paper focuses on experimental evaluation; it presents experiments with real-world web applications which confirm the scalability and precision of the proposed framework and analyses.

7 Conclusions

This paper proposed a practical static analysis framework for inference of security-related program properties. Within the framework, we defined ownership, immutability, and information flow inference analyses. We presented experiments on a set of Java components and on a set of Java web applications. The experiments demonstrated that the analyses are precise and practical and therefore could be incorporated in real-world program comprehension tools.

References

[1] Introduction to Stanford SecuriBench, http://suif.stanford.edu/ livshits/securibench/.

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

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

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

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

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

[7] S. Genaim and F. Spoto. Information flow analysis for Java bytecode. In *VMCAI*, pages 346–362, 2005.

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

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

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

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

[12] Y. Liu and A. Milanova. Static inference of explicit information flow. In *PASTE*, 2008.

[13] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analsysi. Technical report, Stanford University, September 2005.

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

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

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

[17] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1):1–42, 2005.

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

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

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

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

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

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

[24] 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.

[25] 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, Mar. 2001.

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

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

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

[29] 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.

Appendix A

```
   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(cash);
37      register.endSale();
    }
  }
```