Static Analysis for Inference of Explicit Information Flow

Yin Liu
Rensselaer Polytechnic Institute
liuy@cs.rpi.edu

Ana Milanova
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Abstract

This paper proposes a new static analysis for inference of explicit information flow. The analysis is context-sensitive, cubic, and works both on complete programs and software components. We perform experiments on several Java components which show that the analysis is precise and practical. Thus, the analysis can be incorporated in program understanding and verification tools and help verify security properties in a light-weight, practical manner.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms

Keywords points-to analysis, flow analysis

1. Introduction

Unexpected information flow (i.e., violations of the confidentiality or integrity of sensitive program data) can seriously compromise the security and the quality of a software system. Current languages such as Java do not provide effective mechanisms for preventing unexpected information flow.

It is important to study mechanisms for reasoning about information flow and the problem has received considerable attention. The vast majority of work falls into two categories: (1) dynamic, instrumentation-based approaches such as tainting, and (2) static, language-based approaches such as type systems. The disadvantage of the dynamic approaches is that they typically incur significant run-time overhead [8]; 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 [33]; thus, it might be difficult to adopt these approaches in practice. On the other hand, static analysis which works *before program execution* and on *current languages* without the burden of annotations, has received considerably less attention. This is surprising, given that static analysis has great potential to be useful in practice—it does not incur run-time overhead, and it does not require annotations.

Information flow can be classified as *explicit* (information flow that arises explicitly, due to assignment statements), and *implicit* (i.e., flow that arises implicitly, due to conditional statements). This paper proposes a new general-purpose *static* analysis for the inference of *explicit information flow*. This analysis is light-weight, works directly on Java programs before program execution, and does not require annotations by the programmer. It can be incorporated in program understanding and verification tools and help verify in a practical manner the confidentiality and integrity of sensitive program data. We believe that our work, although limited in the sense that it does not handle implicit flow, is a step forward towards the use of static analysis for the purposes of reasoning about information flow; it may help advance the use of static analysis in tools for understanding and verification of security properties.

The analysis is defined as a client of a points-to analysis. Specifically, it uses the well-known Andersen's flow- and context-insensitive points-to analysis for Java [31, 18] which, we conjecture, provides suitable precision for the purposes of information flow inference.The information flow analysis client is flow-insensitive and *context-sensitive*. The context sensitivity scheme is based on CFL-reachability [30, 28]: unlike other popular approaches to context sensitivity such as the function-summary approach and the call-string approach [35], this scheme handles recursion precisely *and* has cubic worst-case complexity. The analysis works both on complete programs and on incomplete programs (i.e., software components). This is an important feature because reasoning about security should be performed on software components; any realistic program understanding and verification tool should be able to work on software components and thus cannot accomodate analysis that works on complete programs. This paper focuses on the analysis of software components which we believe is more relevant and challenging.

We implemented the analysis and performed empirical investigation on several Java components. Specifically, for each component, we examined the fields in component classes in order to determine whether there could be explicit information flow from a field into an untrusted client of the component (i.e., possible violation of confidentiality), and whether there could be explicit information flow from an untrusted client into a field (i.e., possible violation of integrity). We present a precision evaluation which shows that the analysis achieves adequate precision—all identified confidentiality and integrity violations could actually happen for appropriate clients. Furthermore, the analysis is practical—it has cubic worst-case complexity and runs in about 10 seconds on all components. The empirical results indicate that the analysis is precise and practical and therefore can be incorporated in practical software tools for program understanding and verification which will lead to higher quality, more secure software systems.

This work has the following contributions:

- We propose a new static analysis for the inference of explicit information flow. Our analysis is context-sensitive, cubic, and works on both complete programs and software components.
- We present an empirical study on several Java components. It indicates that the analysis is precise and practical.

2. Problem Statement

Run-time Information Flow. 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* [10, 12]. 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 $w \mapsto x$. As stated earlier, this paper considers explicit flows only; handling of implicit flows will be added in the future. For the rest of the paper, the term direct flow refers to direct explicit flow; similarly, information flow refers to explicit information flow.

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 $l_1 = l + x$ 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_3 both point to object o) leads to flow $o.f \mapsto l_4$. Finally $r = l_4 - y$ 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).

Problem Definition. We consider the following information flow analysis problem. 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. Typically, all public classes and their corresponding public methods and public fields are accessible. The classes in *Cls* are *trusted* while the client code built on top of these classes is *untrusted*. Our goal is to design a flow analysis that answers the questions: given a sensitive variable s in *Cls* does there exist client code that exposes information flow from s to some untrusted variable in the client?

Note that this statement addresses the information flow problem of *confidentiality* [33] (i.e., whether sensitive data may flow to untrusted parties). The dual problem of *integrity* (i.e., whether untrusted data may flow to sensitive data) is analogous from the static analysis point of view and therefore, the description in the paper focuses on confidentiality. We have implemented both the confidentiality inference analysis and the integrity inference analysis and present empirical results in Section 4.

Example. Consider package `zip` in Figure 1. 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. 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 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

```
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 1. Sample package `zip`.

```
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 `ph_long`.

Constraints. We employ the following constraint, which is standard for other problem definitions that require analysis of incomplete programs [32, 23]. 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 code that may affect the information flow.

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 Section 4 we included all classes that were transitively referenced by *Cls*. This approach

```
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 2. Placeholder `main` method for `zip`.

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.

3. Information Flow Analysis

The information flow problem outlined in the previous section requires analysis of a partial program. We address this issue by employing a general technique called *fragment analysis* which enables analysis of partial programs; Section 3.1 describes the fragment analysis. Also, the information flow analysis needs points-to information and we employ points-to analysis; Section 3.2 outlines the points-to analysis. The information flow analysis is a client of the points-to analysis; Sections 3.3, 3.4 and 3.5 describe the shallow flow inference and Section 3.6 describes the deep flow inference. Details about the analyses as well as correctness and complexity arguments can be found in [20].

3.1 Fragment Analysis

The analysis 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 [32]. 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 flows that approximate flow over arbitrary clients.

The placeholder `main` method for the classes from Figure 1 is shown in Figure 2. 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; in our example, this includes all public methods in `ZipInputStream` and `ZipEntry`. For details on the fragment analysis see [32].

3.2 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 information flow analysis in two ways: first, aliasing information is needed in order to handle information flow through object fields, and second, call graph information is needed in order to approximate the possible targets at virtual method calls. There is a wide variety of points-to analyses, with different degrees of precision and cost. Our current work uses the well-known flow-insensitive and context-insensitive Andersen's points-to analysis for Java [31, 18].

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 [32].

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

The context-sensitive shallow flow 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 [30], and builds on ideas from [28].

The flow graph $\mathcal{FG}_0$ has two kinds of nodes: variable nodes (e.g., r) and field dereference nodes (e.g., $r.f$). The edges in $\mathcal{FG}_0$ represent direct flows. When building $\mathcal{FG}_0$ the context-sensitive analysis annotates edges with certain information. The summarization and subsequent reachability propagation take these annotations into account and filter out certain infeasible flow paths. Below we describe the processing of each program statement kind.[1]

- **Assignment** $l = (...operator)\ r$ generates flow edge $r \rightsquigarrow l$.
- **Instance field write** $l.f = r$ generates a flow edge $r \overset{*}{\rightsquigarrow} l'.f$ 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 $l'' = l'.f$).
- **Instance field read** $l = r.f$ generates flow edge $r.f \rightsquigarrow l$.
- **Method call** i: $l = r_0.m(r_1, ...)$ generates flow edges $r_0 \overset{(_i}{\rightsquigarrow} \mathit{this}$, $r_1 \overset{(_i}{\rightsquigarrow} p_1$, ... and $\mathit{ret} \overset{)_i}{\rightsquigarrow} l$ for each $m'(\mathit{this}, p_1, ..., \mathit{ret})$ which is a possible run-time target according to the points-to analysis.

There are no annotations on the flow edges generated for assignments and instance field reads. The parentheses annotations at method calls are standard CFL-reachability annotations: they denote flow into context copies of formal parameters, and flow from context copies of return variables. Consider parenthesis $(_i$ in $r_1 \overset{(_i}{\rightsquigarrow} p_1$; it denotes flow from actual parameter r_1 to the instance of the formal parameter p_1 for call site i. Analogously, parenthesis $)_i$ in $\mathit{ret} \overset{)_i}{\rightsquigarrow} l$ 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 handle flow through objects which transcends calling contexts. They 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: $k \overset{(_i}{\rightsquigarrow} p \overset{*}{\rightsquigarrow} \mathit{get.this.f} \rightsquigarrow \mathit{get.ret} \overset{)_j}{\rightsquigarrow} l$. 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. Consider the example in Figures 1 and 2 and let us be interested in the flow of constant `LOCFLG` defined in interface

[1] Notation $\rightsquigarrow$ denotes analysis flow (i.e., the representation of run-time flow), while $\longmapsto$ denotes run-time flow.

`ZipConstants`. The following relevant edges are added to $\mathcal{FG}_0$:

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

3.4 Summarization

Procedure *Summarize* in Figure 3 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. 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:

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

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 `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.off` and `get32.off` $\overset{(_{14}}{\rightsquigarrow}$ `get16.off` are added to worklist WL. Subsequently edge `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.off` is taken off the worklist and processed without the addition of new edges. Edge `get32.off` $\overset{(_{14}}{\rightsquigarrow}$ `get16.off` is taken off the worklist and processed on lines 3-5. The concatenation on line 5 results in new edge `get32.off` $\overset{(_{14}}{\rightsquigarrow}$ `get16.i1` which is added to $\mathcal{FG}^*$ and WL. This edge is then processed on lines 3-5 resulting in new edge `get32.off` $\overset{(_{14}}{\rightsquigarrow}$ `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 `get32.off` $\rightsquigarrow$ `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 `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.off` which was processed on the worklist earlier. The concatenation on line 8 results in new edge `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `get32.i1`. Processing this edge results in new edge `LOCFLG` $\overset{(_6}{\rightsquigarrow}$ `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`.

```
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 3. Computation of shallow flow from s.

3.5 Propagation

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 1 and 2. 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 3) finds nodes v reachable from s; it adds a path edge from s to v to $\mathcal{FG}_p$ with the corresponding flow path annotation. For initialization it considers all edges in $\mathcal{FG}^*$ from s and adds the appropriate path edges to $\mathcal{FG}_p$. Edges of kind $s \overset{(_i}{\rightsquigarrow} v$ result in path edges $s \overset{Call}{\rightsquigarrow} v$. Edges of kinds $s \rightsquigarrow v$, $s \overset{*}{\rightsquigarrow} v$ and $s \overset{)_j}{\rightsquigarrow} v$ result in path edges $s \overset{nCall}{\rightsquigarrow} v$. Each path edge $s \overset{p}{\rightsquigarrow} v_1 \in \mathcal{FG}_p$ is concatenated with edges $v_1 \overset{a}{\rightsquigarrow} v_2 \in \mathcal{FG}^*$. If the concatenation results in a new path edge from s, namely e_3, e_3 is added to $\mathcal{FG}_p$ and WL.

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

The concatenation for *Call* is given below:

$$concat'(s \overset{Call}{\rightsquigarrow} v_1, v_1 \overset{(_i}{\rightsquigarrow} v_2) \quad = \quad s \overset{Call}{\rightsquigarrow} v_2$$

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

Figure 4. Computation of deep flow.

$$concat'(s \overset{Call}{\rightsquigarrow} v_1, v_1 \overset{\text{empty},)_j, *}{\rightsquigarrow} v_2) \quad = \quad \text{NO EDGE!}$$

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:

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

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`:

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

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

3.6 Deep Flow Analysis

Recall that deep flow from s into r occurs if there is a statement $s' = v.f$ such that v points to some object reachable on a sequence of field dereferences from s, and there is shallow flow from s' to r.

Procedure $DeepPropagate$ in Figure 4 states the algorithm for confidentiality inference. It takes as input the summarized flow graph $\mathcal{FG}^*$, a source variable s (i.e., a sensitive variable or field in *Cls*), and a set of untrusted variables $Sinks$ (i.e., all placeholder variables `ph_` from `main`). It uses two auxiliary functions, $FShallow(s)$ and $BShallow(s)$. $FShallow(s)$ returns the set of nodes reachable forward on $\mathcal{FG}^*$ from s, and $BShallow(s)$ returns the set of nodes reachable backwards on $\mathcal{FG}^*$ from s. $FShallow(s)$ and $BShallow(s)$ appropriately handle annotations [20]. The union of $FShallow(s)$ and $BShallow(s)$ gives the set of valid aliases of reference variable s. The output of $DeepPropagate$ is a boolean result: **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.

4. Empirical Results

The static information flow analysis is implemented in Java using the Soot 2.2.3 [39] and Spark [18] frameworks. 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 300MB.

We evaluated the analysis on several Java components from the packages `java.text` and `java.util.zip` (these components were used in related analyses [32] and [23][2]). 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 number of classes in *Cls* 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.

We applied the information flow analysis on the sensitive fields in functionality classes in *Cls*; these include all non-public fields (i.e., fields in *Cls* that are not directly accessible by a client).

Table 2 shows the results. Column (2) shows the number of sensitive fields per component. Columns (3) and (4) show the results of confidentiality inference. Column (3) shows how many of these fields could be leaked to client code through shallow flow according to the analysis, and column (4) shows how many of these fields could be leaked through shallow or deep flow. Columns (5) and (6) show the results of the dual integrity inference analysis. Column (5) shows how many sensitive fields could be tampered by client code through shallow flows according to our analysis, and column (6) shows how many sensitive fields could be tampered through shallow or deep flow.

Analysis precision is of crucial importance for any static analysis. If the information flow analysis is imprecise, it may report a false warning on a sensitive field—i.e., it may report that the field is leaked or tampered, while in fact it is not. False warnings are especially confusing, and a large amount of warnings may prevent the use of the analysis in practice. For example, if the analysis is used in a tool which verifies the security of sensitive data, imprecision will lead to many false warnings. Developers will spend valuable time examining potentially large amount of code until they determine that the warnings are due to analysis imprecision and not to insecure information flow. Clearly, imprecision must be carefully evaluated by analysis designers.

We performed a study of absolute precision [32, 23] on all sensitive fields, a total of 488. Of these, 83 fields were reported as leaked to client code (column (4) in Table 2). We examined manually *each* field s reported as leaked by the analysis and attempted to find a client that would expose information flow from s to a client variable. In each case we successfully constructed such a client. Furthermore, 52 fields were reported as being tampered by client code (column (6) in Table 2). Again, for each field s reported as tampered we attempted to find a client that would expose flow from some client variable to s, and in each case we successfully constructed such a client. Thus, for these components the analysis achieves perfect precision. Note that the analysis is safe in the sense that if a field f is reported as safe, it is guaranteed that there is no explicit flow from f (modulo the constraints outlined in Section 2); however, there could be implicit flow from f since our current analysis does not capture implicit flow.

[2] 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.

(1)Component	(2)Functionality	(3)#Class in *Cls*/ #Functionality	(4)#Fields	(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 of Java components.

(1)Program	(2)#Fields	(3)#Leaked (shallow)	(4)#Leaked (shallow or deep)	(5)#Tempered (shallow)	(6)#Tempered (shallow or deep)	(7) Points-to Analysis	(8) Flow Analysis
`gzip`	15	2(13%)	2(13%)	5(33%)	5(33%)	1m24s	6s
`zip`	29	9(31%)	13(45%)	16(55%)	18(62%)	1m24s	7s
`checked`	3	3(100%)	3(100%)	2(67%)	2(67%)	1m23s	5s
`collator`	134	22(16%)	33(25%)	11(8%)	16(12%)	1m25s	11s
`breaks`	241	6(2%)	7(3%)	5(2%)	5(2%)	1m24s	8s
`number`	66	22(33%)	25(38%)	6(9%)	6(9%)	1m25s	9s

Table 2. Columns 3 and 4: Confidentiality; Columns 5 and 6: Integrity; Columns 7 and 8: Analysis time.

Columns (7) and (8) in Table 2 show the running time of the analysis. Column (7) shows the running time for Soot and Spark, and column (8) shows the running times for the information flow analysis, which includes both confidentiality inference and integrity inference. The analysis performs well on these components.

These results indicate that our static information flow analysis is precise and practical. Therefore, it can be incorporated in program understanding and verification tools and help verify security properties in a light-weight, practical manner.

5. Related Work

Below we discuss related work on static information flow analysis and work on dynamic and language-based approaches. We also discuss work on CFL-reachability-based program flow analysis.

Static information flow analysis. Genaim and Spoto [12] present an information flow analysis for Java bytecode. Unlike our work, their analysis works on complete programs only, and does not separate flow through fields of different objects which may lead to significant imprecision. Sun et al. [37] present an inference flow inference technique; unlike our work, they require summaries of libraries, do not present empirical evaluation and use summary-based context sensitivity. Our analysis is conceptually different from the analyses in [12] and [37]: it is cubic, and based on CFL-reachability which we conjecture, achieves the right scalability and precision for this problem. Also, we present results on absolute precision which indicate that our analysis may achieve better precision. On the other hand, the analyses in [12] and [37] handle implicit flow, while our current analysis does not.

Livshits et al. [21, 17] 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 [21, 17] is exponential (due to the underlying points-to analysis), while ours is cubic.

Shankar et al. [34] describe a related taint analysis for C. Although it supports polymorphic summaries of library functions, the underlying flow analysis is context-insensitive for user functions; in contrast, our analysis is context-sensitive.

Dynamic 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 [25, 27, 41], tools that detect SQL-injection attacks [16, 26, 14], and tools that enforce data confidentiality [6, 38, 13, 22]. Recently, Clause et al. have proposed a general framework for dynamic tainting [8]. 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.

Type-based approaches. These approaches rely on type systems for secure information flow [40, 11, 24, 36, 4, 19]. Generally, they require changes to the language 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. On the other hand, type-based approaches handle implicit flow and generally capture richer concepts (e.g., declassification and delegation) than our analysis.

Semantics-based approach. Semantics-based systems define flow semantics for each language construct and then prove properties related to information flow [2, 3, 5, 1, 7, 15]. Most studies of semantics-based approach focus on simple imperative languages; thus, they do not support object types, aliasing and polymorphism. Although the work in [9] handles some of these features, its applicability in real programs is still unknown. Our static analysis works directly on existing object-oriented languages, supporting all these complex features.

CFL-reachablity. CFL-reachability is a well-known technique for context-sensitive program flow analysis [30]. Our analysis is a CFL-reachability computation: one can easily 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 [28]. Unlike [28], 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 [29], 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).

References

[1] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *Proceedings of Static Analysis Symposium*, pages 100–115, 2004.

[2] G. Andrews and R. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[3] J. Banatre, C. Bryce, and D. M'etayer. Compile-time detection of information flow in sequential programs. In *Proceedings of European Symposium on Research in Computer Security*, pages 55–73, 1994.

[4] 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.

[5] C. Bodei, P. Degano, F. Nielson, and H. Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127:27–41, 2001.

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

[7] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Computer Languages, Systems and Structures*, 28(1):3–28, 2002.

[8] 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.

[9] A. Darvas, R. Hahnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, pages 193–209, 2005.

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

[11] 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.

[12] S. Genaim and F. Spoto. Information flow analysis for Java bytecode. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 346–362, 2005.

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

[14] 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.

[15] R. Joshi and K. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

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

[17] 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.

[18] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

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

[20] Y. Liu and A. Milanova. Static information flow analysis for Java. Technical Report 08-06, Rensselaer Polytechnic Institute, May 2008.

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

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

[23] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 76–85, 2005.

[24] A. Myers. Jflow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[25] 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.

[26] 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.

[27] 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.

[28] J. Rehof and M. Fahndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *ACM Symposium on Principles of Programming Languages*, pages 54–66, 2001.

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

[30] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

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

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

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

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

[35] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

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

[37] 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.

[38] 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.

[39] 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 *International Conference on Compiler Construction*, pages 18–34, 2000.

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

[41] 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.