Static Information Flow Analysis for Java

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

Unexpected information flow can result in vulnerabilities that can compromise the security and availability of software; this can have serious financial, legal and ethical consequences. Current programming languages such as Java do not provide effective mechanisms for preventing unexpected information flow and it is important to develop such mechanisms and advance their usage in software practice.

This paper proposes run-time information flow models, and new static information flow inference analysis. 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.

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. For example, in Java 1.1 the security function `Class.getSigners` mistakenly returned a reference to an internal array; this unexpected flow exposed the array to untrusted clients which could modify the array content and compromise the security of the system. Current languages such as Java do not provide effective mechanisms for preventing unexpected information flow and it is definitely important to develop such mechanisms and advance their usage in software practice.

We propose a new static information flow analysis towards this goal. This analysis is light-weight and works directly on Java programs, before program execution. It can be easily incorporated in program understanding and verification tools and help verify in a light-weight and practical manner the confidentiality and integrity of sensitive program data.

The goal of this work is to i) define a model that captures useful run-time information flow and ii) define a static information flow analysis that infers information flow according to this model. Our model distinguishes two types of flow: *shallow flow* and *deep flow*. Shallow flow covers the standard notion of information flow; that is, there is shallow flow from a variable l into a variable r if changes in the value of l impact the value of r [10]. Deep flow considers flow from (and into) the object structure rooted at a reference variable l; there is deep flow from l into r if a field reachable from l flows to r (i.e., changes in the object structure rooted at l impact the value of r, without changes in the value of l).

This paper proposes a new static information flow inference analysis: the analysis works on existing Java programs and does not require annotations by the programmer. 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 [33, 18] which, we conjecture, provides suitable precision for the purposes of information flow inference.[1] 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 [37], 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.

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 information flow from a field into an untrusted client of the component (i.e., possible violation of confidentiality), and whether there could be 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 run-time model for information flow which

[1] Flow-sensitive analyses distinguish between program points and are more precise and more expensive than flow-insensitive ones. Context-sensitive analyses distinguish between different contexts of invocation of a method and are more precise and typically more expensive than context-insensitive ones.

extends the standard model with the concepts of shallow flow and deep flow.

- We propose a new information flow analysis. 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

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 following 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* (i.e., whether sensitive data may flow to untrusted parties). The description in the paper focuses on confidentiality inference. 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 it is not discussed in detail. Section 5 gives both the confidentiality inference and the integrity inference analysis algorithms; we have implemented both analysis algorithms and present empirical results in Section 6.

Section 2.1 specifies the run-time information semantics and Section 2.2 describes constraints on our model that help achieve more precise and useful analysis.

2.1 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 $w \mapsto x$. For clarity, this paper considers *explicit flows only*; in the future, we plan to extend our work to handle implicit flows as well. For the rest of this paper, the term direct flow is used to refer to direct explicit flow.

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

- **Assignment** $l = (\ldots operator)\ r$ leads to flow $r \mapsto l$.

- **Instance field write** $l.f = r$ leads to flow $r \mapsto o.f$. o is the run-time object referred by l at the point of execution of the statement.

- **Instance field read** $l = r.f$ 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.

- **Method call** $l = r_0.m(r_1, \ldots)$ dispatched to method $m'(this, p_1, \ldots, ret)$ leads to flows $r_0 \mapsto this$, $r_1 \mapsto p_1$, ... and $ret \mapsto l$. Here *this* denotes the implicit parameter `this` of method m', p_i denote the formal parameters of m', and *ret* denotes a special variable that holds the return value of m'.

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

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

Again, the problem is the following: given a variable s in component *Cls* and the above definition of information flow, does there exist a client that would expose information flow from s into some variable v in the client code?

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. First, consider local variable `e` in method `readLOC`. It is easy to see that one can write a client of this component which exposes shallow flow from `e` to the client.[2] For example, consider the following client:

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

Let o_{ZIS} stand for the run-time `ZipInputStream` object created in the above client. The shallow flow results as follows: `readLOC.e` $\mapsto$ `readLOC.ret` due to line 10, followed by `readLOC.ret` $\mapsto$ `getNextEntry.e` due to line 2, followed by `getNextEntry.e` $\mapsto$ o_{ZIS}`.entry` due to line 3, followed by o_{ZIS}`.entry` $\mapsto$ `getNextEntry.ret` due to line 4, and finally `getNextEntry.ret` $\mapsto$ `ph_ZE` due to the call to `getNextEntry` in the client. Second, consider flow from field `tmpbuf` in `ZipInputStream`. Let o_{ZE} stand for the run-time `ZipEntry` object created at line 5. One cannot write a client which exposes shallow flow from `tmpbuf` (i.e., the reference to the array `tmpbuf` is never exposed to a client). However, one

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

can write a client which exposes deep flow from `tmpbuf` (i.e., the content of the array is exposed). Consider client

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

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

2.2 Discussion

We employ the following constraint, which is standard for other problem definitions that require analysis of incomplete programs [34, 32, 22]. We only consider executions in which the invocation of a boundary method does not leave *Cls*—that is, all of its transitive callees are also in *Cls*. If we consider the possibility of unknown subclasses, all instance calls from *Cls* could be "redirected" to unknown external code that may affect the information flow inference. For example, a field may be confidential in the current set of classes but an unknown subclass may override a method and leak the field (e.g., by using the confidential field in a computation of the value of a public static field).

Thus, *Cls* is augmented to include the classes that provide component functionality as well as all other classes transitively referenced. In the experiments presented in Section 6 we included all classes that were transitively referenced by *Cls*. This approach restricts analysis information to the currently "known world"—that is, the information may be invalidated in the future when new subclasses are added to *Cls*. One could change the analysis to make worst case assumptions for calls that may enter unknown methods; however, in this case the analysis will be overly conservative and will likely report less useful information.

3. OVERVIEW OF 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; the fragment analysis is described in Section 3.1. Furthermore, the information flow analysis needs points-to information and we employ points-to analysis; the points-to analysis is outlined in Section 3.2. The information flow analysis is defined as a client of the points-to analysis. Section 3.3 describes an intuitive context-insensitive information flow analysis; this analysis is the foundation for the analyses presented in Sections 4 and 5.

3.1 Fragment Analysis

Clearly, the problem considered in this paper requires analysis of a partial program. The input is a set of classes *Cls* and the analysis needs to approximate information flow that is valid across all possible executions of arbitrary client code built on top of *Cls*. To address this problem we make use of a general technique called *fragment analysis* due to Nasko Rountev [31, 34]. Fragment analysis works on a pro-

```
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.

```
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`.

gram 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 [31, 34].

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 known and relatively well-understood Andersen's points-to analysis [33, 18]. 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).

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 [31, 34].

3.3 Context-insensitive Information Flow Analysis

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

The flow graph represents direct flows. There are two kinds of nodes: variable nodes (e.g., r) and field dereference nodes (e.g., $r.f$). Consequently, there are three kinds of direct flow edges: (1) $l \rightsquigarrow r$ which represents flow from variable l into variable r, (2) $l \rightsquigarrow r.f$, which represents flow from variable l into field f of an object referred to by r, and (3) $l.f \rightsquigarrow r$ which represents flow from field f of an object referred to by l into variable r. We use notation $\rightsquigarrow$ to distinguish from notation $\mapsto$; $\rightsquigarrow$ denotes analysis flow (i.e., the representation of run-time flow), while $\mapsto$ denotes run-time flow. Below we describe the processing of each program statement kind.

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

The handling of assignments and method calls directly follows the information flow semantics defined in Section 2.1. The handling of field accesses uses alias information instead of points-to information; we believe that this representation is more informative when tracking indirect information flow.

Tracking shallow flow from a source variable s amounts to a straightforward reachability computation: while there are new edges and we have not reached an untrusted variable we examine pairs $s \rightsquigarrow v_1 \rightsquigarrow v_2$ and add $s \rightsquigarrow v_2$ to the flow graph if $s \rightsquigarrow v_2$ is not already there.

Tracking deep flow from s requires tracking shallow flow from multiple sources. There is a worklist of sources which is initialized to s. While there are sources on the worklist, the analysis takes a source s' and performs shallow flow computation from s'. For each s' of reference type, the analysis finds all aliases v of s' (i.e., nodes reachable forward and backwards through shallow flow from s') and examines each field read statement $s'' = v.f$; if s'' is a new source, it is added to the worklist of sources. Each s'' (or s') is part of the object structure rooted at s; subsequently, the shallow flow computation from s'' checks for violating flow from s''.

Note that the information flow analysis, as described above is again a whole-program analysis. As with the points-to analysis the placeholder `main` enables the use of the whole-program information flow analysis. The `main` method approximates all possible clients that could be built on top of *Cls* and thus the result of the whole-program information flow analysis includes all flows that could result from individual clients (clearly, under the constraints in Section 2.2).

4. CONTEXT-SENSITIVE SHALLOW FLOW ANALYSIS

The context-insensitive analysis is bound to produce substantial imprecision, as well as overhead in analysis cost due to the tracking of infeasible flow. Therefore, there is a need for a context-sensitive analysis—that is, analysis that tracks flow through different contexts of invocation of a method precisely. Section 4.1 illustrates the imprecision of context-insensitive information flow analysis, and Sections 4.2, 4.3 and 4.4 describe in detail the context-sensitive information flow analysis. Section 4.5 discusses the termination, complexity and correctness properties of the analysis.

4.1 Imprecision of Context-insensitive Analysis

Example. Consider the example in Figures 1 and 2 and let us be interested in the flow of constant `LOCFLG` defined in interface `ZipConstants`. The following edges created by the context-insensitive analysis represent flow relevant to `LOCFLG`:

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

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

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

The context-sensitive analysis consists of three parts: generation of flow graph $\mathcal{FG}_0$, summarization of the effects of callee onto callers, and demand-driven reachability propagation on the summarized graph. This analysis is based on CFL-reachability, and is most closely related to work by Reps et al. [30] and Rehof and Fahndrich [28].

When building $\mathcal{FG}_0$ the context-sensitive analysis annotates edges with certain information. The summarization and subsequent reachability propagation take these annotations into account and filter out infeasible paths.

There are no annotations on the flow edges generated for assignments and instance field reads; these are treated as in Section 3.3. Instance field writes and method calls are annotated as follows:

- **Instance field write** $l.f = r$ generates a flow edge $r \overset{*}{\rightsquigarrow} l'.f$ for every variable l' aliased with l such that there is a read of f from 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 run-time target method $m'(\mathit{this}, p_1, ..., \mathit{ret})$.

The parentheses annotations at method calls are standard CFL-reachability annotations: they denote flow into context copies of formal parameters, and flow from context copies of return variables. Consider parenthesis $(_i$ in $r_1 \overset{(_i}{\rightsquigarrow} p_1$; it denotes flow from actual parameter r_1 to the instance of the formal parameter p_1 for call site i. Analogously, parenthesis $)_i$ in $\mathit{ret} \overset{)_i}{\rightsquigarrow} l$ denotes flow from the instance of return variable ret for call site i to the left-hand side of the call l. The parentheses are matched to form valid flow paths — for example, $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow \mathit{ret} \overset{)_i}{\rightsquigarrow} l_1$ is a valid path, but $i_1 \overset{(_i}{\rightsquigarrow} p_1 \rightsquigarrow \mathit{ret} \overset{)_j}{\rightsquigarrow} l_2$ is not.

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

Example. Let us continue with the edges from the previous section. With context-sensitive analysis the edges will have annotations as follows:

`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`)

4.3 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. There are two cases. Lines 3-5 process the case when the annotation on edge e_1 is of kind $(_i$. In this case, the algorithm examines each edge e_2 which is a successor of e_1 and concatenates e_1 and e_2. If this concatenation results in a new edge e_3, e_3 is added to $\mathcal{FG}^*$ and WL. Lines 6-8 process the case when the annotation on edge e_1 is empty or $*$. The algorithm examines each predecessor edge e_2', already added to $\mathcal{FG}^*$ and WL, and possibly taken off WL before e_1. It concatenates e_2' with e_1 and if this concatenation results in a new edge e_3', e_3' is added to $\mathcal{FG}^*$ and WL. It is important to note that operation *concat* produces an edge *only if* the first edge has a $(_i$ annotation, and the second edge has one of the following annotations: empty, $*$, or matching $)_i$; otherwise, there is no edge:

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

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 $\overset{(_6}{\leadsto}$ get32.off and get32.off $\overset{(_{14}}{\leadsto}$ get16.off are added to worklist *WL*. Subsequently edge LOCFLG $\overset{(_6}{\leadsto}$ get32.off is taken off the worklist and processed without the addition of new edges. Edge get32.off $\overset{(_{14}}{\leadsto}$ get16.off is taken off the worklist and processed on lines 3-5. The concatenation on line 5 results in new edge

get32.off $\overset{(_{14}}{\leadsto}$ 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 $\overset{(_{14}}{\leadsto}$ 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 $\leadsto$ 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 $\overset{(_6}{\leadsto}$ get32.off which was processed on the worklist earlier. The concatenation on line 8 results in new edge

LOCFLG $\overset{(_6}{\leadsto}$ get32.i1.

Processing this edge results in new edge

LOCFLG $\overset{(_6}{\leadsto}$ get32.ret.

Processing this edge does not result in new edges. Edges LOCFLG $\overset{(_6}{\leadsto}$ get32.ret and get32.ret $\overset{)_8}{\leadsto}$ 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.

4.4 Propagation

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

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

The *Call* annotation denotes flow paths that end on a call sequence. In our example, there is a *Call* path LOCFLG $\overset{Call}{\leadsto}$ 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.

```
procedure **Summarize**
input      $\mathcal{FG}_0$: flow graph
output     $\mathcal{FG}^*$: summarized $\mathcal{FG}_0$
initialize $\mathcal{FG}^* = \mathcal{FG}_0$
           $WL = \{v_1 \overset{a}{\leadsto} v_2 \in \mathcal{FG}_0$ s.t. $a$ is $(_i$ annotation$\}$
[1] while $WL \neq \emptyset$ do
[2]   remove $e_1$: $v_1 \overset{a_1}{\leadsto} v_2$ from $WL$
[3]   if $a_1$ is an $(_i$ annotation
[4]     foreach $e_2$: $v_2 \overset{a_2}{\leadsto} v_3 \in \mathcal{FG}^*$ 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 $e_2'$: $v_0 \overset{a_2'}{\leadsto} v_1 \in \mathcal{FG}^*$ 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 $e_1$: $s \overset{p}{\leadsto} v_1$ from $WL$
[3]     foreach $e_2$: $v_1 \overset{a}{\leadsto} v_2 \in \mathcal{FG}^*$ 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 source node s.

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 $\overset{nCall}{\leadsto}$ getSize.ret which is due to flow get32.i1 $\overset{)_8}{\leadsto}$ readLOC.i2, followed by flow readLOC.i2 $\overset{*}{\leadsto}$ getSize.this.size, followed by flow getSize.this.size $\leadsto$ 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 $s \overset{(_i}{\leadsto} v$ result in path edges $s \overset{Call}{\leadsto} v$. Edges of kinds $s \leadsto v$, $s \overset{*}{\leadsto} v$ and $s \overset{)_j}{\leadsto} v$ result in path edges $s \overset{nCall}{\leadsto} v$. Each path edge $s \overset{p}{\leadsto} v_1 \in \mathcal{FG}_p$ is concatenated with edges $v_1 \overset{a}{\leadsto} v_2 \in \mathcal{FG}^*$. If the concatenation results in a new path edge from s, namely e_3, e_3 is added to $\mathcal{FG}_p$ and *WL*.

It remains to define the concatenation operation *concat'*. We need to consider concatenation of each possible path annotation (i.e., (1) *Call* and (2) *nCall*), with each possible edge annotation (i.e., (1) empty, (2) $(_i$, (3) $)_j$ and (4) $*$). A path annotation is preserved by an empty edge annotation—that is, we have $concat'(s \overset{p}{\leadsto} v_1, v_1 \leadsto v_2) = s \overset{p}{\leadsto} v_2$. A path annotation concatenated with $(_i$ results in *Call*—that is, we have $concat'(s \overset{p}{\leadsto} v_1, v_1 \overset{(_i}{\leadsto} v_2) = s \overset{Call}{\leadsto} v_2$. The concatenation for $)_j$ is given below:

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

In the first case, the *nCall* path annotations is preserved—clearly, adding a return edge still results in a path that does not end on a call sequence. In the second case, no edges are added: 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 edge $)_j$ is accounted for in *Summarize*.

Finally, the concatenation for $*$ is given below:

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

The *nCall* path is preserved, and no edges are added to *Call* paths—again, the relevant flow has been computed in *Summarize*.

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.

4.5 Termination, Complexity and Correctness

Termination. Consider procedure *Summarize* in Figure 3. There is finite number of new $(_{...}$-annotated edges, finite number of new $*$-annotated edges, and finite number of new empty edges. Therefore $\mathcal{FG}^*$ reaches a fixed point and *Summarize* terminates. Consider procedure *Propagate*. For every node $v \in \mathcal{FG}^*$ there are at most 2 path edges between s and v and therefore $\mathcal{FG}_p$ reaches a fixed point and *Propagate* terminates as well.

Complexity. Let N be the size of the program being analyzed—that is, the number of statements, the number of methods and the number of variables is of order N. For each pair of nodes v_i, $v_j \in \mathcal{FG}^*$ there could be at most 3 edges between them: (1) a $*$-annotated edge, (2) an empty edge, or (3) *one* of a $(_{...}$ edge or a $)_{...}$ edge. Thus there are at most $O(N^2)$ edges that are processed on the worklist *WL* in *Summarize*. For each edge the algorithm does at most $O(N)$ work examining successor edges (lines 3-5), or examining predecessor edges (lines 6-8). Therefore, the complexity of *Summarize* is $O(N^3)$. Similarly, in *Propagate*, for a given source s, there are only 2 possible path edges between s and a node v, thus only $O(N)$ paths are processed on *WL*. For each edge the algorithm does at most $O(N)$ work examining successor edges (lines 3-4), thus it is easy to see that *Propagate* has complexity of $O(N^2)$. Therefore, the complexity of computing flow for all sources is $O(N^3)$.

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

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

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

Sketch of proof. The proof is by induction on the depth of the stack from v_1. For depth of 1 we have path $v_1 \overset{call}{\mapsto} g_1 \overset{ret}{\mapsto} v_2$ where g_1 is a sequence of variable-to-variable intraprocedural flows. Therefore the flow sequence has the form: $v_1 \overset{call}{\mapsto} p \mapsto w_1 \mapsto w_2 \ldots w_n \mapsto ret \overset{ret}{\mapsto} v_2$ where p is the instance of the formal parameter, ret is the instance of the return variable, and w_i are the instances of the local variables for stack frame g_1. This path has analysis representative $v_1 \overset{(_1}{\rightsquigarrow} p \rightsquigarrow w_1 \rightsquigarrow w_2 \rightsquigarrow \ldots w_n \rightsquigarrow ret \overset{)_1}{\rightsquigarrow} v_2$, where 1 is the index of the call site that triggers the invocation of stack frame g_1. It is easy to see that annotation $(_1$ will be propagated through variables p and w_i (lines 3-5 in *Summarize*), resulting in an edge $v_1 \overset{(_1}{\rightsquigarrow} ret$, which will be concatenated with $ret \overset{)_1}{\rightsquigarrow} v_2$ resulting in the needed edge $v_1 \rightsquigarrow v_2$.

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

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

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

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

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

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

5. DEEP FLOW ANALYSIS

This section considers computation of *deep flow*. Section 5.1 presents the algorithm for deep flow computation for the purpose of detecting confidentiality violations. Section 5.2 presents the dual algorithm for deep flow computation for the purpose of detecting integrity violations.

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

The algorithms use two auxiliary functions, *FShallow*(s) and *BShallow*(s), where s is an arbitrary node in the flow graph $\mathcal{FG}^*$. Function *FShallow*(s) returns the set of nodes reachable forward through shallow flow from s. This set is computed by *Propagate* in Figure 3. Analogously, function *BShallow*(s) returns the set of nodes reachable *backwards* through shallow flow from s; in other words, it contains nodes v such that there is shallow flow from v to s. The set is computed analogously to *Propagate*, only backwards: the algorithm keeps path edges e_1: $v_1 \overset{p}{\rightsquigarrow} s$ on the worklist and examines edges e_2: $v_2 \overset{a}{\rightsquigarrow} v_1$ from $\mathcal{FG}^*$; subsequently, e_2 is concatenated with e_1 and the resulting edge e_3 (if any) is added to the path graph $\mathcal{FG}_p^{-1}$ and to the worklist.

```
procedure DeepPropagate
input       $\mathcal{FG}^*$: summarized flow graph
            $s$: source node    Sinks: sink 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 $SLW$
[8] return true;
```

Figure 4: Computation of deep flow from all sources.

Note that one needs different path annotations for backward reachability. Again, there are two kinds of path annotations: $Call^{-1}$, and $nCall^{-1}$. $Call^{-1}$ denotes flow paths that begin with a call sequence. $nCall^{-1}$ denotes paths that do not begin with a call sequence. These paths could be: (1) empty paths, (2) paths that begin with a return sequence, or (3) paths that begin with $*$. Analogously to forward propagation, we need to consider concatenation of each possible edge annotation (i.e., (1) empty, (2) $(_i$, (3) $)_j$, and (4) $*$) with each possible path annotation (i.e., (1) $Call^{-1}$, and (2) $nCall^{-1}$). The empty edge annotation preserves the path annotation—that is, we have $concat''(v_2 \rightsquigarrow v_1, v_1 \overset{p}{\rightsquigarrow} s) = v_2 \overset{p}{\rightsquigarrow} s$. The $)_j$ annotation always results in an $nCall^{-1}$ path—that is, we have $concat''(v_2 \overset{)_j}{\rightsquigarrow} v_1, v_1 \overset{p}{\rightsquigarrow} s) = v_2 \overset{nCall^{-1}}{\rightsquigarrow} s$. Similarly, the $*$ annotation always results in an $nCall^{-1}$ as well: $concat''(v_2 \overset{*}{\rightsquigarrow} v_1, v_1 \overset{p}{\rightsquigarrow} s) = v_2 \overset{nCall^{-1}}{\rightsquigarrow} s$. Finally, the concatenation for $(_i$ is given below:

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

In the first case, the $(_i$ annotation preserves the $Call^{-1}$ path. In the second case no new path is needed as the $(_i$ annotation has been propagated forward during *Summarize*.

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

5.1 Confidentiality Inference

Recall from Section 2 that if a sensitive variable s is a reference variable, there may be flow from the object structure rooted at s. By definition (Section 2), 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 l, and there is shallow flow from s' to r.

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

Example. Recall the example of deep flow from Section 2.1. Source node s is `tmpbuf` and *Sinks* includes all

(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.

placeholder variables from `main` in Figure 2. Initially `tmpbuf` is added to *SWL*. Then `tmpbuf` is taken off the worklist and we have $FShallow(\texttt{tmpbuf}) = \{\texttt{get32.b}, \texttt{get16.b}\}$. This set does not intersect with *Sinks* and the analysis proceeds. Set $BShallow(\texttt{tmpbuf})$ is empty and lines 5-6 in the algorithm examine only `tmpbuf`, `get32.b`, and `get16.b` for indirect reads. There is indirect read from `get16.b` on line 11 in Figure 1 and `get16.b1` is added to *SWL*. The algorithm proceeds to compute $FShallow(\texttt{get16.b1})$ which includes `ph_long` from `main`; therefore, the result is **false**.

It remains to show that this algorithm actually computes deep flow as defined in Section 2.1 and reiterated in the beginning of this section. Intuitively, one needs to show that every relevant indirect field read $s' = v.f$ is examined by our algorithm. This is done by considering induction on the length of the field path. Assume, that each field read $s' = v.f_k$ such that v is aliased with $s.f_1...f_{k-1}$ is examined (recall that s is the source sensitive variable). We need to show that every field read $s'' = v'.f_{k+1}$ such that v' is aliased with $s.f_1...f_{k-1}.f_k$, is examined as well. Our analysis assumes that each object field is read at least once—that is, there is at least one read statement $s' = v.f_k$ which reads $s.f_1...f_{k-1}$. Then v' and s' are aliased since they refer to the same object, namely the one referred to by $s.f_1...f_{k_1}.f_k$. By the inductive hypothesis, statement $s' = v.f_k$ is examined and therefore s' is processed on the worklist and sets $FShallow(s')$ and $BSshallow(s')$ are computed. v' must be included in one of these two sets (either there is shallow flow from s' to v' and $v' \in FShallow(s')$ or there is shallow flow from v' to s' and $v' \in BShallow(s')$). Therefore, statement $s'' = v'.f_{k+1}$ is examined as well.

5.2 Integrity Inference

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

The integrity inference is the dual of the confidentiality inference in Figure 4. It has the same inputs as the algorithm for confidentiality inference and differs only slightly (the two adjustments are highlighted in bold):

```
[3] if $\mathbf{BShallow(s)} \cap Sinks \neq \emptyset$ return false
[4] if $s$ is of reference type
[5]   foreach $v \in \{s\} \cup FShallow(s) \cup BShallow(s)$ do
[6]    foreach indirect **write** $\mathbf{v.f = s'}$ do
[7]     add $s'$ to $SWL$
```

Line 3 considers set $BShallow(s)$—that is, the set of variables r such that there is shallow flow from r into s. If some of these variables is in *Sinks* the algorithm returns false. Lines 4-7 are needed for deep flow computation. Line 5-6 find all valid aliases v of s and examine field writes $v.f = s'$. Each s' is part of the object structure rooted at s; it is put on the worklist and subsequently, line 3 checks for violating shallow flow into s'.

6. EMPIRICAL RESULTS

The empirical study aims to address two questions. First, how *imprecise* the analysis is—that is, how often it reports safe fields as compromised due to insecure information flow? Second, does the analysis have good performance?

The static information flow analysis is implemented in Java using the Soot 2.2.3 [42] 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 [22][3]). 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.

6.1 Results

We applied the information flow inference analyses from Section 5 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 of the confidentiality inference analysis (Section 5.1). Column #Fields shows the number of sensitive fields per component. Column #Leaked(shallow) shows how many of these fields could be leaked to client code through shallow flow according to the analysis. Column #Leaked(shallow or deep) shows how many of these fields could be leaked through shallow or deep flow.

Table 3 shows the results of the dual integrity inference analysis. Column #Tampered(shallow) shows how many sensitive fields could be tampered by client code through

[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	#Leaked (shallow)	#Leaked (shallow or deep)
gzip	15	2(13.33%)	2(13.33%)
zip	29	9(31.03%)	13(44.83%)
checked	3	3(100%)	3(100%)
collator	134	22(16.42%)	33(24.63%)
breaks	241	6(2.49%)	7(2.90%)
number	66	22(33.33%)	25(37.88%)

Table 2: Confidentiality: fields leaked to client code.

shallow flows according to our analysis. Column #Tampered(shallow or deep) shows how many sensitive fields could be tampered through shallow flow or through deep flow.

Program	#Fields	#Tampered (shallow)	#Tampered (shallow or deep)
gzip	15	5(33.33%)	5(33.33%)
zip	29	16(55.17%)	18(62.07%)
checked	3	2(66.67%)	2(66.67%)
collator	134	11(8.21%)	16(11.94%)
breaks	241	5(2.07%)	5(2.07%)
number	66	6(9.09%)	6(9.09%)

Table 3: Integrity: fields tampered by client code.

Tables 2 and 3 show that deep flow does capture additional, potentially insecure flow. Essentially, it captures flows that leak or tamper sensitive data *partially* (e.g., an element of an internal container is leaked or tampered, while the reference to the container itself remains secure). Clearly, the larger and more complex a component, the more likely it is that deep flow captures additional flow (e.g., `collator` and `breaks`). We conjecture that reasoning about deep flow is necessary—it is likely to capture more potentially insecure flow, especially in larger and more complex components.

Program	Points-to Analysis	Flow Analysis
gzip	1m24s	6s
zip	1m24s	7s
checked	1m23s	5s
collator	1m25s	11s
breaks	1m24s	8s
number	1m25s	9s

Table 4: Analysis time.

Table 4 shows the the running time of the analysis. The first column shows the running time for Soot and Spark, and the next column shows the running times for the information flow analysis, which includes both confidentiality inference and integrity inference. Clearly, the analysis performs well on these components—it runs in only several seconds, and its cost is a small fraction of the underlying points-to analysis.

6.2 Analysis Precision

The issue of 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 [34, 22] on all sensitive fields, a total of 488 fields. Of these fields, 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 4 in Table 3). 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. It is important to note that the analysis is safe—that is, all fields reported as safe, are in fact safe (modulo the constraints outlined in Section 2.2).

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.

7. RELATED WORK

Section 7.1 considers work on secure information flow, and Section 7.2 considers work on CFL-reachability-based program flow analysis.

7.1 Secure Information Flow

There is a large amount of work on secure information flow. The vast majority of it 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, compiler and run-time system [35]; thus, it would be difficult to adopt these approaches in current software practice. On the other hand, static analysis which works *before program execution* and on *existing languages* 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 changes to the language or user input in the form of annotations. We believe that our work is a step forward in this important direction; it may help advance the use of static analysis in real-world tools for understanding and verification of security properties.

Below we discuss directly related work on static information flow analysis, and work on dynamic and language-based approaches.

Static information flow analysis. Genaim and Spoto [12] present an information flow analysis for Java bytecode. Their analysis works on complete programs only, and does not separate flow through fields of different objects which may lead to significant imprecision; in contrast, our analysis works both on complete and incomplete programs and separates flow through different object fields. Furthermore, our analysis is conceptually different: it is cubic, and based on CFL-reachability which we conjecture, achieves the right scalability and precision for this problem. Finally, we present results on absolute precision which indicate that our analysis may achieve better precision.

Livshits et al. [20, 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 [20, 17] is exponential (due to the underlying points-to analysis), while ours is cubic.

Related type inference techniques have been proposed [36, 4, 40]. One disadvantage of these techniques is that they lack support for libraries: they either require users to provide type annotations for libraries, or restrict the usage of libraries. In contrast, our analysis handles libraries seemlessly: it analyzes reachable library code and tracks flow through this code.

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 [24, 27, 44], tools that detect SQL-injection attacks [16, 25, 14], and tools that enforce data confidentiality [6, 41, 13, 21]. 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 [43, 11, 23, 38, 19]. Generally, these approaches require changes to the language, compiler and run-time system, as well as sometimes complex type annotations provided by the programmer; therefore, it may be difficult to adopt these approaches in practice. In contrast, our analysis works directly on Java codes and does not require annotations; it can be directly incorporated in program understanding and verification tools.

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.

7.2 CFL-reachability

CFL-reachability is a well-known technique for context-sensitive program flow analysis [30]; it has been used in a variety of flow analyses that require context sensitivity (e.g., points-to analysis for Java [39], and analysis for race detection for C [26]). Our analysis is a CFL-reachability computation as well; one can see that the *concat* operations are essentially grammar productions. We conjecture that CFL-reachability presents the right degree of scalability and precision for the problem of static information flow analysis.

Our work builds on the ideas in [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).

8. CONCLUSIONS

In this paper, we have proposed a new static information flow analysis. The contributions of our work are the following: first, we define a run-time information flow model that extends the standard model with the concepts of *shallow flow* and *deep flow*. Second, we present a new static information flow analysis that infers information flow accordingly. The analysis works on complete and incomplete Java programs, is context-sensitive and has cubic worst-case complexity. Finally, we present an empirical investigation on several Java components that indicates that our analysis is practical and precise.

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

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

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

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

[24] 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.

[25] 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.

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

[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. *Dataflow Analysis of Software Fragments.* PhD thesis, Rutgers University, 2002.

[32] A. Rountev. Precise identification of side-effect free methods. In *IEEE International Conference on Software Maintenance*, pages 82–91, 2004.

[33] 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*, pages 43–55, 2001.

[34] 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.

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

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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.