# Static Information Flow Analysis with Handling of Implicit Flows and A Study on Effects of Explicit Flows vs Implicit Flows

Yin Liu
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th St., Troy, NY 12180, USA
liuy@cs.rpi.edu

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th St., Troy, NY 12180, USA
milanova@cs.rpi.edu

## Abstract

*Reasoning about information flow can help software engineering. Static information flow inference analysis is a technique which automatically infers information flows based on data or control dependence. It can be utilized for the purposes of general program understanding, detection of security attacks and security vulnerabilities, and type inference for security type systems.*

*This paper proposes a new static information flow inference analysis, which unlike most other information flow analyses, handles both explicit and implicit information flows. The analysis does not require annotations and it is relatively precise and practical.*

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

*We study the impact of implicit flow versus explicit flow in these client applications. Implicit flow has significant impact on all these applications. In security violation detection, implicit flow detects more security violations than explicit flow. In type inference, implicit flow infers more untrusted type variables. In the study of the effect of thread-shared variables, implicit flow detects more affected variables than explicit flow.*

## 1 Introduction

It is important to study mechanisms for reasoning about information flow and the problem has received considerable attention. The vast majority work on information flow 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 [4]; 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 [26]; 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.

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

For security violation detection, information flow inference can help detect security attacks such as script, web cache, web page, and SQL injection attacks; it can help detect security vulnerabilities that release confidential data to untrustworthy parties (e.g., echoing a password on the screen). Security type systems such as JFlow [18], require that variables and statements are annotated with security types (labels which denote security levels). Type inference based on information flow inference analysis, can lessen the burden of manually writing type annotations, and can greatly facilitate the application of security type systems in

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

This paper proposes a new general-purpose *static* information flow analysis. Unlike other recent static flow analyses which consider only explicit flows [14, 29, 31], it has a novel analysis that captures both explicit flows and *implicit flows* (i.e., flows due to conditional statements). Our analysis is light-weight, works directly on Java programs before program execution, and does not require annotations by the programmer. The analysis is defined as a client of a points-to analysis. The information flow analysis client is flow-insensitive and *context-sensitive*[1].

We implemented the static information flow analysis and performed empirical investigation. We illustrate the analysis on three client applications. The first client application is security violation detection. We perform experiments on a suite of Java web applications. Specifically, we identify a set of *untrusted* data (i.e., data that can potentially carry malicious data that can be injected into the application)[2]. We also identify a set of *sensitive* data (i.e., data that manipulates the application)[3]. For each program, we examine whether there could be information flow from *untrusted* data to *sensitive* data (i.e. whether the application could be attacked by the injected malicious data). In our experiments, our analysis effectively detects a total of 26 real security vulnerabilities which do actually compromise the security of the applications.

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

The last client application studies the effect of thread-shared variables on thread-local variables. We perform experiments on a set of multi-thread programs. Specifically, we identify a set of *thread-shared* variables and a set of *thread-local* variables. For each program, we study how many thread-local variables could be reached from thread-shared variables due to information flow (i.e. how many thread-local variables are affected by thread-shared variables). In our experiments, at least 1/3 of the thread-local variables are affected by thread-shared variables.

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

The analysis is practical for all the programs applied. It runs in about 1 to 4 minutes on most benchmarks, and costs less than 9 minutes on the largest benchmark. The empirical results on these applications indicate that the analysis is precise and practical and therefore can be incorporated in practical software tools for program understanding and verification.

This work has the following contributions:

- We propose a context-sensitive information flow inference analysis that captures both explicit and *implicit* information flow.
- We illustrate the usage of our static flow analysis on three client applications: security violation detection, type inference, and a study of the effect of thread-shared variables on thread-local variables.
- We study the impact of implicit flow vs. explicit flow on each client application. The experiments show that implicit flow has significant impact.
- We present an empirical study on a suite of Java web applications and multi-threaded programs, which demonstrates that the analysis is adequately precise and practical.

## 2 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 val-

[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.

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

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

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

**Figure 1. Guess-a-Number web application**

ues of $x$ are observable from the output values of $y$. Such flows are *direct* and *indirect* [5, 7]. Direct flows can be *explicit* (i.e., data-flow based) and *implicit* (i.e., control-flow based). Direct explicit flows arise at assignment statements, while direct implicit flows arise from conditionals. Indirect (i.e., transitive) flows arise from compositions of direct flows: e.g. for the sequence of statements `z=x+5; if(z>1) y=5;`, the direct explicit flow $x \mapsto z$ and direct implicit flow $z \mapsto y$ lead to indirect flow $x \mapsto y$. This paper considers both *explicit* flows and *implicit* flows.

As an illustrating example, consider a simple web application in which the user has a few chances to guess a number between 1 and 10, and wins if a guess is correct. This example is taken from [2]. Figure 1 shows a key fragment of the source code of the application. Field `secret` is the true number (the number that needs to be guessed), field `tries` shows the number of guesses taken by the user, and method `makeGuess` contains the core logic of the application. Methods `finishApp` and `message.setText` print out their String parameters, and are designated as untrusted outputs (i.e., outputs that make information visible to the untrusted user).

In this application, information flow analysis can help reason about the following data confidentiality requirement: the untrusted user should not learn the true number, until a guess is correct. This requirement is expressed more formally as follows: there should be no information flow from `secret` to any untrusted output — the print statements at lines 10, 14, 16 and 17. The analysis easily infers that there is no explicit flow from `secret` to untrusted output. The analysis also infers that there is implicit flow from `secret` to the outputs at lines 10, 14 and 16 — clearly, the value of `secret` is checked by the conditional at line 8 and therefore, the outputs at lines 10, 14 and 16 reveal information about the value of `secret`. This information flow happens to be benign and is permitted; in a security type system such as JFlow, permission must be granted explicitly through *declassification*.

## 3 Information Flow Analysis

The information flow analysis needs points-to information and we employ points-to analysis; Section 3.1 outlines the points-to analysis. Sections 3.2 and 3.3 describe the information flow inference analysis.

### 3.1 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 [25, 12].

### 3.2 Construction of Flow Graph $\mathcal{FG}_0$

The context-sensitive flow analysis consists of three parts: generation of *flow graph* $\mathcal{FG}_0$, summarization of the effects of callees onto callers, and demand-driven reachability propagation on the summarized graph. This analysis is based on CFL-reachability [24], and builds on ideas from [22].

The flow graph $\mathcal{FG}_0$ has several kinds of nodes for capturing explicit and implicit flows: variable nodes (e.g., $r$), field dereference nodes (e.g., $r.f$), method nodes (e.g., $m$ that break down inter-procedural implicit flow into intra-procedural flows), and conditional statement nodes (e.g., $s$ that break down implicit flows which cross nested conditionals). The edges in $\mathcal{FG}_0$ represent flows that could not be broken down into smaller flows.

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

**Figure 2. Construction of flow graph.**

The construction of flow graph $\mathcal{FG}_0$ is illustrated in Figure 2. When building $\mathcal{FG}_0$ the context-sensitive analysis annotates edges with certain information. The summarization and subsequent reachability propagation take these annotations into account and filter out certain infeasible flow paths. Below we describe the flow edges constructed in this stage.[4]

### 3.2.1 Explicit Flow Edges

The explicit flow edges are straight-forward. For each direct assignment, instance field write, and instance field read statement[5], the data is transmitted from the right-hand-side expression variables to the left-hand-side variable. For example, for direct assignments $l = r$, there is explicit flow edge $r \rightsquigarrow l$. Flows through fields need to be captured in a special way in order to account for aliasing. For example, flows $l_1 \rightsquigarrow v_1.f$ and $v_2.f \rightsquigarrow l_2$ where $v_1$ and $v_2$ are aliases (i.e., they could point to the same object at run-time), lead to flow $l_1 \rightsquigarrow l_2$. Therefore, whenever a field write $l.f = r$ occurs, explicit flow edges are generated between the right-hand-side $r$ and all written object fields $l'.f$, where $l'$ is an alias of $l$. Instead of building an edge $r \overset{*}{\rightsquigarrow} l'.f$, we break it into edges $r \overset{*}{\rightsquigarrow} l.f$, and $l.f \rightsquigarrow l'.f$. This is done in order to enable a precision improvement which is explained in Section 3.3. For method call statement, each run-time method invocation has the effect of generating explicit flow from actual arguments to the corresponding instances of the formal parameters, as well as generating explicit flow from the instance of the return variable to the variable on the left-hand-side of the method call.

### 3.2.2 Implicit Flow Edges

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

For each conditional statement, a node $s_i$ is created to capture the flows through this conditional statement. For example, in Figure 1, node $s_8$ represents the *if* statement at line 8 and node $s_7$ represents the *if* statement at line 7. For each variable tested at $s_8$, we generate implicit flow edges. There are implicit flow edges $\texttt{tries} \rightsquigarrow s_8$, $\texttt{i} \rightsquigarrow s_8$ and $\texttt{secret} \rightsquigarrow s_8$. These edges illustrate that flow transmitted through statement $s_8$ transfers the information from variables `tries`, `i` and `secret`.

For each conditional statement $s_i$, we check if there exists conditional statement $s$ immediately enclosing $s_i$. If there exists such a statement, we build an implicit flow edge from $s$ to $s_l$. For example, in Figure 1, $s_8$ at line 8 has an immediately enclosing, namely $s_7$, the *if* statement at line 7. The implicit flow edge $s_7 \rightsquigarrow s_8$ shows that flow through $s_8$ carries information from $s_7$ as well. Thus, the implicit flow across nested conditionals $s_7$ and $s_8$, such as the flow $\texttt{i} \rightsquigarrow \texttt{tries}$, would be captured by connecting several flows: $\texttt{i} \rightsquigarrow s_7$ (when processing statement at line 7), $s_7 \rightsquigarrow s_8$ (what we just described), and later another implicit flow $s_8 \rightsquigarrow \texttt{tries}$ (as it will be described below).

For each assignment, instance field read, and instance field write, we check if there exists a conditional statement $s$ immediately enclosing the assignment. If there exist such a conditional, we build implicit flow edge from $s$ to left-hand-side variable of the assignment. In Figure 1, the assignment at line 9 has an immediately enclosing *if* statement, $s_8$ at line 8. An implicit flow edge $s_8 \rightsquigarrow \texttt{tries}$ shows that the assignment of variable `tries` is affected by the control transfer at $s_8$. Similarly, the implicit flow $\texttt{secret} \rightsquigarrow \texttt{tries}$ would be captured by connecting flow edges $\texttt{secret} \rightsquigarrow s_8$ (when processing line 8) and

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

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

$s_8 \rightsquigarrow$ `tries` (described here).

Method call statements are processed similarly to assignment statements. Each formal parameter $p_i$ is considered as assigned. Thus, implicit flow edges are constructed connecting the immediately enclosing conditional $s$ and $p_i$. Another implicit flow edge, $s \rightsquigarrow m'$ is constructed, to capture the inter-procedural flow from the current method to the target method $m'$. In Figure 1, line 14 calls method `message.setText`; this call is affected by the *if* statement at line 13, and thus we have $s_{13} \rightsquigarrow m_{setText}$ which illustrates that flow transmitted to method $m_{setText}$ transfers the information from conditional $s_{13}$.

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

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

The assignment at line 19 has no immediately enclosing conditional statement in `setText`. However, method `message.setText` is called at line 14 in Figure 1, and therefore it is affected by the conditional statement $s_{13}$. Thus, the flow effect of $s_{13}$ on variable `text` needs to be captured. At line 19, implicit flow edge $m_{setText} \rightsquigarrow$ `text` is constructed, which connects with the edge $s_{13} \rightsquigarrow m_{setText}$ to capture the inter-procedural flow $s_{13} \rightsquigarrow$ `text`.

#### 3.2.3 Conditional Scope

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

As inter-procedural implicit flow is broken down into intra-procedural flow, the scope of a conditional statement could be identified intra-procedurally. The scope identification is based on the intra-procedural control flow graph(CFG).[6] Each node in the graph represents a basic block, i.e. several statements without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow.

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

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

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 $(_i\}$
[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. Summarization and Propagation.**

#### 3.2.4 Flow Edge Annotations

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

The $*$ annotations at field writes handle flow through ob-

jects 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 $r$ to the value of $k$ (i.e., there is statement `this.f=p;` in method `set`). Later there is a call `j: l=r.get()` which returns field $f$ of $r$ (i.e., there is statement `return this.f;`). The flow edges for these statements are: $k \overset{(i}{\rightsquigarrow} p \overset{*}{\rightsquigarrow} get.this.f \rightsquigarrow get.ret \overset{)_j}{\rightsquigarrow} l$. In the above example $(_i$ is concatenated with the wildcard and it is "cancelled" by it resulting in transitive flow edge $k \overset{*}{\rightsquigarrow} get.ret$. Subsequently, the wildcard "cancels" $)_j$ resulting in flow edge $k \overset{*}{\rightsquigarrow} l$.

### 3.3 Summarization and Propagation

Procedure *Summarize* in Figure 3 computes the summary flow graph $\mathcal{FG}^*$. Intuitively, this procedure computes the flow effects due to method calls, propagating $(_i$ annotations forward until they are matched with a corresponding $)_i$ or a $*$ annotation. It is important to note that operation *concat* produces an edge *only if* the first edge has a $(_i$ annotation, and the second edge has one of the following annotations: empty, $*$, or matching $)_i$; otherwise, there is no edge:

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

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.

To improve precision, we separate the handling of $*$-annotated edges to $\texttt{this}.f$. When a call context $(_i$ is cancelled by such a $*$ annotation, $\texttt{this}.f$ is replaced by $v'.f$ s.t. $v'$ is alias of $v$, the receiver of the method call. Here the call context helps to precisely capture the points-to information of the field references, which introduces object sensitivity that improves analysis precision.

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 $s$: $Call$, denotes flow paths that end on a call sequence; and $nCall$, denotes paths that do not end on a call sequence.

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 $\mathcal{FG}_p$. 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 $WL$.

The concatenation for path edges is given below:

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

## 4 Applications of Information Flow Analysis

To illustrate the usage of our static information flow analysis, we implemented our analysis and applied it on several applications.

The static information flow analysis is implemented in Java using the Soot 2.2.3 [33] and Spark [12] 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 1000MB.

### 4.1 Security Violation Detection

For security violation, our analysis is performed on a set of web applications, SecuriBench [16], established as security benchmarks[7]. `jboard`, `blueblog`, `personalblog`, `pebble`, and `roller` are web-based bulletin board and blogging applications. `webgoat` is a J2EE application designed as a test case and a teaching tool for Web application security.

The benchmarks are described in the first three columns of Table 1. The version of each benchmark is shown in Column (2). The possible *sources* for security attacks are untrusted inputs such as fields including hidden fields of HTML forms, submitted URLs, received HTTP requests, obtained cookies, etc (identified according to [16]). The number of these sources is shown in Column (3). Sensitive data for possible security attacks are variables that must be trusted, such as parameters passed to particular

[7]The set consists of 8 applications. However, application `blojsom` from the suite is no longer available; we included all other applications except for one, `snipsnap`, which did not run through our Soot-based infrastructure.

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

**Table 1.** Security violations in Web application security benchmarks.

methods as SQL query,written values of server-side output streams,commands executed by the system,file paths or values sent to http response. The number of sensitive variables is shown in Column (4) (identified according to [16]). The Column (5) shows the number of methods, including library methods, determined to be reachable by Spark.

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

We examine the impact of implicit flow on this application. First, we run the version of the analysis which creates $\mathcal{FG}_0$ with explicit flow edges only. The number of detected security violations is shown in the first sub-column of Column (6) in Table 1. Next, we run the version analysis which creates $\mathcal{FG}_0$ with both explicit and implicit flow edges. The number of detected violations is shown in the second sub-column of Column(6). Implicit flow has no impact on four benchmarks (i.e., it does not detect additional security violations compared to explicit flow). However, on *blueblog* and *webgoat*, it detects 7 additional security violations per benchmark. Implicit flow captures more security violations; our examination confirms that the reported violations are not false positives.

We compared the results of our explicit flow analysis with the results reported by the explicit flow analysis in [16]. Our results are the same as reported in [16], except for 2 cases. In application `webgoat`, we discovered 3 more security violations than what was reported in [16]; our manual examination confirmed these security violations. Second, our analysis did not discover the 2 violations in application `personalblog`.

Overall, the precision experiments confirm that our inexpensive analysis achieves very good results.

### 4.2 Type Inference

Security type systems such as JFlow [18], require that variables and statements are annotated with security types (labels which denote security levels). Type inference based on information flow inference analysis, can lessen the burden of manually writing type annotations, and can greatly facilitate the application of security type systems in software practice.

We perform type inference on the same set of security benchmarks described in Section 4.1. The security type being inferred is the *untrusted* type. The initial set of *untrusted* type variables are variables identified as *sources* describes in Section 4.1.

For each program, the analysis propagates the *untrusted* type labels from the initial set. This essentially is a taint analysis — it identifies flow from the untrusted *sources* specified in Table 1, to all other variables in the program. The number of inferred untrusted type variables is shown in Table 2.

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

Column (4) in Table 2 shows the combined running time for security violation detection and type inference. The running time illustrates that the analysis is practical – it runs within 1 to 4 minutes for most benchmarks, except for `roller` on which it runs in around 8 minutes.

### 4.3 Effect of Thread-shared Variables

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

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

**Table 2. Type inference for untrusted variables.**

| (1)Benchmark | (2)#Reachable Methods | (3)#Thread -shared | (4)#Effected Thread-local | | (5)#Effected Fields | | (6)#Time |
|---|---|---|---|---|---|---|---|
| | | | Explicit | Incl. Implicit | Explicit | Incl. Implicit | |
| hedc | 4403 | 36 | 420 (31.8%) | 912 (69.1%) | 30 (51.7%) | 58 (100%) | 24s |
| sor | 3578 | 4 | 154 (69.4%) | 212 (95.5%) | 4 (40%) | 4 (40%) | 16s |
| tsp | 3415 | 15 | 254 (74.5%) | 323 (94.7%) | 9 (75%) | 11 (91.7%) | 16s |
| montecarlo | 3526 | 17 | 204 (71.3%) | 267 (93.3%) | 36 (87.8%) | 39 (95.1%) | 14s |
| raytracer | 3481 | 4 | 201 (36.7%) | 380 (69.3%) | 4 (8.33%) | 20 (41.7%) | 14s |

**Table 3. Effects of thread-shared variables.**

(montecarlo and raytracer), and programs from ETH (a Traveling Salesman Problem solver tsp, a successive over-relaxation benchmark sor, and a web crawler hedc). Column (2) in Table 3 shows the number of reachable methods for each benchmark, including library methods, determined by Spark.

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

Our analysis examines the effect of thread-shared variables by tracking information flows from these thread-shared variables. If there exists information flow from a thread-shared variable to a thread-local variable, we say that the thread-local variable is affected by the thread-shared variable; this states that the value of the thread-local variable is dependent on that thread-shared variable. The number and percentage of affected thread-local variables are shown in Column (4) of Table 3. The number and percentage of affected written instance fields are shown in Column (5). These affected variables and fields may need to be examined carefully, because they depend on the values of thread-shared variables.

As with the other applications, we examine the impact of implicit flow. First, we run the version of analysis which creates $\mathcal{FG}_0$ with explicit flow edges only. The first sub-column of Column (4) in Table 3 shows that on average 1/3 of the thread-local variables are effected by thread-shared variables. The first sub-column of Column (5) shows that on average 1/2 of the written instance fields are effected by thread-shared variables. Second, we run the version of the analysis which considers both explicit and implicit flow. The results show that implicit flow impacts significantly this application. The second sub-columns of Column (4) and Column (5) show that implicit flow captures at least 20% additional affected thread-local variables and 10 to 40% additional affected written fields.

Column (6) in Table 3 shows the running time of the analysis on each program. The analysis runs within 30 seconds on all the benchmarks.

## 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 [7] present an information flow analysis for Java bytecode. Their analysis does not separate flow through fields of different objects which may lead to significant imprecision; in contrast, our analysis separates flow through different object fields. Furthermore, our analysis is conceptually different: it is 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. [16, 11] propose analysis for finding vulnerabilities caused by unchecked inputs. This analysis only tracks flow of objects, while our analysis considers flow for both object and simple types. Furthermore, their analysis

considers only explicit flows, while ours tracks both explicit and implicit flows. Again, our analysis is conceptually different: the analysis in [16, 11] is exponential (due to the underlying points-to analysis), while ours is cubic.

Tripp et al. [31] propose a static taint analysis for Java that detects security vulnerabilities by reasoning about information flows. They first construct a context-sensitive points-to analysis, and then use a hybrid algorithm for slice construction to track tainted data. They focus on explicit flows only, not considering information flows through control dependencies, while our analysis considers both explicit and implicit flows.

Our previous work on information flow [14] presents a static analysis for inferring explicit information flows. Its application on security violation detection is presented in [15]. Unlike our previous work which considers explicit flow only, this new work has following new contributions: first, it deals with implicit flows; second, it provides several client applications of information flow analysis; last, it studies the impact of implicit flow.

Related type inference techniques have been proposed [27, 1, 30]. 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 [19, 21, 35], tools that detect SQL-injection attacks [10, 20, 9], and tools that enforce data confidentiality [3, 32, 8, 17]. Recently, Clause et al. have proposed a general framework for dynamic tainting [4]. 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 [34, 6, 18, 28, 13]. Generally, they 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.

**CFL-reachablity.** CFL-reachability is a well-known technique for context-sensitive program flow analysis [24]. 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 [22]. Unlike [22], 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 [23], 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).

## 6 Conclusions

The contributions of our work are the following: first, we present a new static information flow analysis. This analysis captures both explicit and *implicit* information flows. It is context-sensitive and has cubic worst-case complexity. Second, we illustrate the usage of this static flow analysis on three client applications. One client application is security violation detection, another is type inference, and the last application studies the effect of thread-shared variables on thread-local variables. Third, we study the impact of implicit flow on each client application. The experimental results show that implicit flow has significant — that is, it could detect additional security violations, infers additional untrusted type variables, and detects additional affected thread-local variables. Last, our empirical investigation on a Java web application suite and a multi-threaded application suite indicates that our analysis is practical and precise.

## References


[1] 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.

[2] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 31–44, 2007.

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

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

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

[14] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 50–56, 2008.

[15] Y. Liu and A. Milanova. Practical static analysis for inference of security-related program properties. In *IEEE International Conference on Program Comprehension*, pages 50–59, 2009.

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

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

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

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

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

[29] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

[30] 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.

[31] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page to appear, 2009.

[32] 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.

[33] 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.

[34] 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.

[35] 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.