CFL-Reachability and Context-sensitive Integrity Types

Ana Milanova   Wei Huang   Yao Dong
Rensselaer Polytechnic Institute
Troy, NY, USA
{milanova, huangw5, dongy6}@cs.rpi.edu

Abstract
Integrity types can help detect information flow vulnerabilities in web applications and Android apps. We study SFlow, a context-sensitive integrity type system and we give an interpretation of SFlow in terms of CFL-reachability. We propose SFlowCFL, a new, more precise integrity type system, and SFlowCFL-Infer, the corresponding type inference analysis, which is equivalent to CFL-reachability. SFlowCFL-Infer is an effective taint analysis for Android. It scales well and detects numerous privacy leaks in popular Android apps.

1. Introduction
Pluggable types enhance traditional types. They help find bugs or verify the absence of bugs. With the addition of JSR 308 [6] to Java 8, pluggable types will be increasingly important.

One important class of pluggable types, which we term information flow types, helps support information flow control. The type qualifiers and subtyping hierarchy are:

\[ \text{neg} <: \text{poly} <: \text{pos} \]

Here neg is the “negative” qualifier and pos is the “positive” qualifier. The goal of the type system is to ensure that there is no flow from a “positive” source to a “negative” sink. Poly is a polymorphic qualifier, which is interpreted as pos in some contexts, and as neg in other contexts.

There are many examples of information flow types. Reference immutability types [13, 32] prevent flow from a readonly variable \( x \) (readonly is the “positive” qualifier) to a mutable variable \( y \) (mutable is the “negative” qualifier, e.g., \( y.f = 2 \)). This ensures that a readonly reference \( x \) is never used to mutate the referenced object. Energy types [28] prevent flow from an approximate variable \( x \), which can be represented energy-efficiently, to a precise variable \( y \).

One class of information flow types, which we term integrity types, tackles traditional integrity and confidentiality violations [1, 7, 14, 15, 30]. Integrity types prevent flow from untrusted tainted sources (tainted is the “positive” qualifier) to sensitive safe sinks (safe is the “negative” qualifier). They can help uncover violations such as SQL-injection and cross-site scripting (XSS), or verify the absence of such violations. Furthermore, integrity types can help uncover leaks of private data (e.g., phone and location data) to untrusted parties (e.g., Internet) in Android apps, or they can verify the absence of such leaks. For the rest of the paper, we focus on integrity types, although our techniques apply to information flow types in general.

Intuitively, the integrity type system propagates safe sinks backwards through the program. If safe propagates to a variable marked as tainted, then there is a type error signaling potential flow from a tainted source to a safe sink; otherwise, the type system guarantees that there is no flow from a source to a sink. Just as with traditional types, integrity types require annotations, which may hinder practical adoption. Therefore, type inference is important.

We advocate type-based integrity analysis, which consists of an integrity type system and the corresponding type inference. This analysis is modular and compositional. It can analyze any given set of classes \( P \). Unknown callees in \( P \) are handled using appropriate defaults. Callers of \( P \) can be analyzed separately and composed with \( P \) without reanalysis of \( P \). Our type-based approach handles reflective object creation seamlessly. This is possible because the type-based analysis does not require abstraction of heap objects; instead, it models flow from one variable to another through subtyping. The analysis requires annotations only on sources and sinks. Once the sources and sinks are built into annotated libraries, code is analyzed without any input from the user. There is evidence that our type-based approach outperforms traditional points-to and dataflow-based integrity analysis [14, 15]. Given these advantages and the advance

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied] ...$10.00

---

1“Integrity types” is clearly a misnomer because in Android the concern is confidentiality. “Confidentiality types” and qualifiers secret and public would have been more appropriate terms. We keep the term “integrity types” and qualifiers tainted and safe in deference to previous work, which consistently has used the term “taint analysis” [2, 5].
of JSR 308, it is important to study type-based integrity analysis and its connection to established program analysis techniques.

In this paper, we build upon our recent work on the context-sensitive integrity type system SFlow [14, 15]. We study the connection of SFlow to Context-free Language (CFL)-reachability, which is a well-established program analysis technique [25–27]. We proceed to define SFlowCFL, a novel more precise integrity type system, and SFlowCFL-Infer, an inference analysis equivalent to CFL-reachability.

The paper makes these concrete contributions:

- A novel, CFL-reachability-based integrity analysis.
- SFlowCFL, a new context-sensitive integrity type systems.
- Interpretation of SFlow and SFlowCFL in terms of CFL-reachability.
- SFlowCFL-Infer, an inference analysis equivalent to CFL-reachability. SFlowCFL-Infer is an effective taint analysis for Android.

The rest of the paper is organized as follows. Sect. 2 presents the CFL-reachability-based integrity analysis. Sect. 3 presents the SFlow type system. Sect. 4 presents SFlowCFL and SFlowCFL-Infer, the corresponding inference analysis. Sect. 5 explains the handling of inheritance and virtual calls. Sect. 6 discusses the application of SFlowCFL-Infer to the detection of privacy leaks in Android apps. Sect. 7 presents related work and Sect. 8 concludes.

2. CFL-reachability-based Integrity Analysis

This section begins with an overview of CFL-reachability (Sect. 2.1). It proceeds to define the CFL-reachability-based integrity analysis (Sect. 2.2 and Sect. 2.3).

2.1 Overview of CFL-reachability

Context Free Language (CFL)-reachability is an established program analysis technique [25–27]. CFL-reachability casts the problem as a reachability problem over a directed graph G. The nodes in the graph correspond to program variables and the edges represent the flows between these variables.

The following is a classic example of CFL-reachability:

\[
\begin{align*}
C &::= P N | P | N \\
P &::= M | \{x(i,M)|M\} | rP \\
M &::= d | \{r|M\} | M M
\end{align*}
\]

(a) Context-free grammar C

(Fig. 1(a) shows grammar C, which handles call-transmitted dependences. \(i\) ranges over all callsites. (b) F handles structure-transmitted dependences. \(f\) ranges over all fields. \(L(C)\) and \(L(F)\) denote the languages generated by C and F respectively.

5, a flows to p. Finally, flow from the return value to the left-hand-side of the call assignment is represented with \(\rightarrow\)-annotated edges, where, again, \(i\) is the corresponding callsite.

CFL-reachability casts the problem as follows: if there is a path from \(x\) to \(y\) in \(G\), such that the annotations on this path form a string in the context-free language of balanced parentheses, then \(x\) flows to \(y\). If the annotations do not form a string in this language, \(x\) does not flow to \(y\). In the id example above, \(a\) flows to \(b\) because the string \([d_5d_6]\) belongs to the language. However, \(a\) does not flow to \(d\) because \([d_5d_6]\) does not belong to the language. The intuition is clear: CFL-reachability avoids spurious flow across different contexts of invocation, because it matches each call with the corresponding return. The above example illustrates the handling of call-transmitted dependences [27] (i.e., dependences due to flow through method calls).

Fig. 1(a) shows grammar C, which handles call-transmitted dependences. \(M\)-paths are paths of balanced (matched) parentheses that represent intra-procedural flows. In the id example, \(d\) and \([d_5d_6]\) are \(M\)-paths. \(N\)-paths represent paths with outstanding calls, e.g., the path from a to \(b\) has string \([d_5d]\) which is generated by \(N\). \(P\)-paths represent paths due to returns, e.g., the path from \(a\) to \(b\) has string \([d_5d]\) generated by \(P\). We abuse notation slightly by having \(C\) denote both the grammar and the starting non-terminal.

Java requires handling of structure-transmitted dependences [27] (i.e., dependences due to flow through fields):

\[
\begin{align*}
1 & x.f = a; \\
2 & x.g = c; \\
3 & y = x; \\
4 & b = y.f; \\
5 & d = y.g.
\end{align*}
\]

Above, \([r]\)-annotated edges model field writes and \([r]\)-annotated edges model field reads. The edge from \(a\) to \(x\) represents the fact that \(a\) flows to \(f\) of \(x\) (due to \(x.f = a\) at line 1). Structure-transmitted dependences can be modeled using an analogous context-free grammar, shown in Fig. 1(b).

Here \(a\) flows to \(b\) because string \([r d]\) belongs to the language generated by the grammar in Fig. 1(b). However, \(a\) does not flow to \(d\) because \([r d]\) does not belong to the language.
Had there been only call-transmitted or only structure-transmitted dependences, CFL-reachability could be solved in time $O(n^3)$, where $n$ is the size of the program [25]. Unfortunately, there could be interleaving of the two kinds of flow dependences:

1. $X \text{get}_f() \{ \text{return this.$f$;} \}
2. $x = a$;
3. $b = x.$get$_f()$

Field $f$ of $x$ is written before the call to $get_f$, it is retrieved in $get_f$ and returned into $b$, creating the path from $a$ to $b$ with interleaved parentheses and brackets. Reps has shown that the precise handling of interleaved parentheses and brackets, i.e., the precise handling of both call-transmitted and structure-transmitted dependences is undecidable [27].

There are additional complications. So far, we create edges as follows: (1) an edge from left-hand-side $x$ to left-hand-side $x$ at direct assignment $x = y$, (2) an edge from actual to formal, (3) an edge from return value to left-hand-side of call assignment, (4) an edge from $y$ to receiver $x$ at field write $x.f = y$, and (5) an edge from receiver $w$ to $z$ at field read $z = w.f$. Consider the code in Fig. 2(a). The call at line 12 to set produces two $(12)$-annotated edges: $x \xrightarrow{(12)} \text{this$_s$}$ and $a \xrightarrow{(12)} p$. If we were to use only those edges, the flow from $a$ to $b$ would not be discovered! The problem is that this$_s$ becomes mutated. The value written into field $f$ of this$_s$ is in effect returned to the caller of set; the “return” must be taken into account.

To overcome this problem we make use of reference immutability [13, 32]. Specifically, if the left-hand side of an assignment (explicit or implicit) is immutable, then there is one edge, in the expected direction. Otherwise, there is an edge in the expected direction and an additional inverse edge in the opposite direction, reversing the annotation as well.

At line 12 in Fig. 2, there is one edge due to the implicit assignment of actual $a$ to formal $p$: $a \xrightarrow{(12)} p$. This edge suffices because $p$ is immutable. However, there are two edges due to the implicit assignment of $x$ to this$_s$: $x \xrightarrow{(12)} \text{this$_s$}$ and inverse edge this$_s \xrightarrow{(12)} x$. Note that the annotation is reversed as well, $(12)$ becomes $(12)$; this expresses the fact that the value written into $f$ of this$_s$ is returned to the caller of set. The inverse edge is added because there is mutation of this$_s$ at this.$f = p$. With the inverse edge, there is a path from $a$ to $b$ with string $(12 \xrightarrow{(t)} 12 \xrightarrow{(d)} 14 \xrightarrow{(d)} 14 \xrightarrow{(d)} r)$ as shown in Fig. 2(b). Parentheses and brackets both match.

The intuition behind inverse edges is the following. If there is a write into field $f$ of some $y$, each path $x \rightarrow \cdots \rightarrow y$ must be reversed, because field $f$ can be read out of $x$, or out of any $z$ to which $x$ flows. Consider

$w = x$;
$y = x$;
$y.f = z$;
$v = w.f$;

In this example, slightly modified from [31], $y.f$ and $w.f$ are aliases and $z$ flows to $v$. The inverse edge $y \rightarrow x$ is added because $y$ is modified at $y.f = z$. This edge connects $y$ to $x$, and then to $w$, which reveals the flow from $z$ to $v$.

Inverse edges and inverse paths are similar to the ones in Sridharan and Bodik [31]. The key novelty in our work is the use of reference immutability, which obviates the need for heap abstraction. Sridharan and Bodik [31] require heap abstraction and hence they incur the notorious difficulties of dealing with reflective object creation.
The syntax is shown in Fig. 3. Without loss of generality, we may assume that each field annotation is transmitted dependences. We write $x \xrightarrow{c|f} y$ where $c$ is the call annotation and $f$ is the field annotation. The type qualifiers present in the figure become relevant in Sect. 3.

2.2 Algorithm

This section formalizes the intuition into a CFL-reachability-based integrity analysis. We define the analysis over a syntax in "named form" where the results of field accesses, method calls, and instantiations are immediately stored in a variable. The syntax is shown in Fig. 3. Without loss of generality, we may assume that methods have parameter this, and exactly one other formal parameter. The type qualifiers present in the figure become relevant in Sect. 3.

The analysis separates the grammar of call-transmitted dependences from the grammar of structure-transmitted dependences. This is achieved by having each edge in the graph present with two orthogonal annotations, one that tracks call-transmitted dependences and another one that tracks structure-transmitted dependences. We write $x \xrightarrow{c|f} y$. $c$ is the call annotation; it is a string in $L(C)$, or a nonterminal in $C$ representing a string in $L(C)$. $f$ is the field annotation; it is a string in $L(F)$, or a nonterminal in $F$.

The analysis, called CFL-Solver, is shown in Fig. 4. It takes as input a set of classes $P$ and a sink variable $n$. (Without loss of generality we assume that there is a single sink $n$.) CFL-Solver propagates $n$ through $P$; it builds the graph "backwards", i.e., in opposite direction of the flow edges, adding nodes and edges as it discovers paths to $n$. CFL-Solver outputs a directed graph $G$ that safely approximates flows to $n$. If there is flow from reference $x$ to $n$, then $x$ is in $G$ and $n$ is reachable from $x$. If $x$ is tainted, CFL-Solver signals potential flow from a source to a sink. For clarity, at this point we assume that every virtual call has a single target. We discuss inheritance and virtual calls in Sect. 5.

CFL-Solver makes use of ReIm, a type system for reference immutability we developed in previous work [13]. ReIm comes with a quadratic type inference analysis. If ReIm infers that a reference $x$ is readable, this guarantees that $x$ is never used to mutate the referenced object nor anything it references. If $x$ is readable, all of the following are forbidden:

- $x.f = z$
- $x.setField(z)$ where setField sets a field of its receiver
- $y = \text{Id}(x)$; $y.f = z$
- $x.f.g = z$
- $y = x.f$; $y.g = z$

Function $\text{ReIm}(expr)$ returns the ReIm type of the expression argument. Procedure $\text{BiEdge}(lhs, s, t, c, f)$, shown at lines 21-24, adds edges due to explicit and implicit assignments. It takes $lhs$ — the left-hand-side of the assignment, $s$ — the source and target of the edge respectively, and $c$ and $f$ — the call and field annotations of the new edges. CFL-Solver calls $\text{BiEdge}$ at every program statement (lines 4-12). If the left-hand-side of the assignment is readable according to ReIm, $\text{BiEdge}$ adds only one edge, from $s$ to $t$. Otherwise, $\text{BiEdge}$ adds an inverse edge from $t$ to $s$ as well. $\tau$ and $\overline{\tau}$ are as expected: $\tau = |t|$, $\overline{\tau} = |t|$, $\overline{\tau} = |t|$, $\overline{\tau} = |t|$, $\overline{\tau} = |t|$. As an example, consider $\text{WRITE} x.f = x$. If left-hand-side $y.f$ is readable, then $\text{BiEdge}$ creates edges $x \xrightarrow{y.f} y$. Otherwise, it creates edges $x \xrightarrow{y.f}$ and $y \xrightarrow{y.f} x$. Fig. 2(b) shows the edges added due to lines 4-12. (There are details we clarify shortly.)

Lines 13-14 add edges that represent $M$-paths, that is, paths of balanced parentheses. To reduce clutter we use $d$ instead of $M$. For example, line 13 takes edge $x \xrightarrow{d|f} y$ and edge $y \xrightarrow{d|f} z$ and adds a new edge $x \xrightarrow{d|f\oplus f'} z$, where $f\oplus f'$ is the concatenation of the two field annotations $f$ and $f'$. Lines 15-18 add $P$-paths, $N$-paths and $C$-paths (recall Fig. 1(a)).

Consider Fig. 2 and edges $y \xrightarrow{d|f} \text{this}_{get} \xrightarrow{d|d} \text{ret}_{get}$. Line 15 adds edge $\text{this}_{get} \xrightarrow{N|d} \text{ret}_{get}$ due to production $N ::= M$. Lines 16-18 then add edge $y \xrightarrow{N|d} \text{ret}_{get}$ due to production $N ::= (\_ N$, recording the $N$-path from $y$ to $\text{ret}_{get}$. Fig. 2(c) shows $P$-paths, $N$-paths and $C$-paths.

Note that CFL-Solver collapses the strings in $L(C)$ into representative nonterminals (thus eschewing explicit comments)

---

2 Grammar $C$ can be written into Chomsky normal form, which will render lines 13-14 unnecessary. We separate the handling of $M$-paths (lines 13-14) from the handling of the rest of the paths (15-18) to better illustrate the parallel with type system $SFlow$. 
putation of these strings). However, it explicitly computes strings in $L(F)$ (of course, subject to approximation as explained shortly). This means that we have chosen to handle call-transmitted dependences precisely but approximate in the handling of structure-transmitted dependences.

Procedure EDGE (lines 25-29) adds an edge from source $s$ to target $t$ with call annotation $c$ and field annotation $f$. It adds the edge only if the following conditions are met: (1) sink $n$ is reachable from target $t$ on a path in $L(C)$, (2) the field annotation $f$ is in $L(F)$ and (3) if $f$ is a field write $\lf$, the corresponding field read $\rf$ must be in $G$. The rule of condition (3) becomes clear in the following subsection.

### 2.3 Approximation

As stated earlier, the precise handling of both call-transmitted and structure-transmitted dependences is undecidable. Therefore, analyses must approximate in the handling of one or both of these dependences. Analyses typically approximate in the handling of call-transmitted dependences [31].

In contrast to previous work, our analysis handles call-transmitted dependences precisely but approximates in the handling of structure-transmitted dependences. Recall that CFL-Solver was designed to explain SFlow. SFlow as well as other information flow type systems [13, 28] approximate in the handling of structure-transmitted dependences. Our approximation is simple: $f \oplus f'$ always evaluates to $d$, thus erasing field annotations. Just as any approximation, this approximation may introduce imprecision.

Condition (3) in EDGE mitigates the imprecision. If field $f$ never flows to sink $n$, then $\rf$ is never added to $G$ and the sink does not propagate towards writes of $f$. Consider

```plaintext
1. x.f = a;
2. x.g = c;
3. y = x;
4. b = y.f;
5. safe d = y.g; // d is sink variable n
```

Only field $g$ is added to the graph, $f$ is not added because $b$ does not flow to the sink. Thus, the safe sink is propagated backwards to $c$ as expected, but it is not propagated to $a$. Imprecision arises only when there are two instances of the same class, say $x$ and $y$, and two distinct fields, say $x.f$ and $y.g$ flow to the sink. The sink is propagated backwards towards $x.g$ and $y.f$ even though these fields may not flow to the sink. In our experience with both Java web applications and Android apps, this is rarely a source of imprecision, because sources and sinks are sparse.

CFL-Solver is in fact a framework for CFL-reachability analysis. Analysis designers can choose a different approximation for fields and encode this approximation through grammar $F$ and concatenation operation $\oplus$. For example, one can reduce $F$ to a regular grammar just as one can reduce $C$ to a regular grammar [31]. Furthermore, one can handle structure-transmitted dependences precisely but approximate in the handling of call-transmitted dependences. We plan to explore these directions in future work.

From now on, when referring to CFL-Solver, we mean the solver implementing the above approximation for fields.

## 3. Type-based Integrity Analysis

This section describes the type-based integrity analysis. We outline the SFlow type system and interpret it in terms of the CFL-reachability analysis.

### 3.1 Type Qualifiers

Each variable is typed by a type qualifier (Fig. 3). There are two basic qualifiers in SFlow: tainted and safe.

- **tainted**: A variable $x$ is tainted, if there is flow from a source to $x$. Sources are tainted.
- **safe**: A variable $x$ is safe if there is flow from $x$ to a sink. Sinks are safe.

In order to disallow flow from tainted sources to safe sinks, SFlow enforces the following subtyping hierarchy:

```
safe <: tainted
d```

where $q_1 <: q_2$ denotes $q_1$ is a subtype of $q_2$ ($q$ is also a subtype of itself $q <: q$). Therefore, it is allowed to assign a safe variable to a tainted one:

```plaintext
safe String s = ...;
safe String t = s;
```

However, it is not allowed to assign a tainted variable to a safe one:

```plaintext
tainted String t = ...;
safe String s = t; // type error!
```

### 3.2 Context Sensitivity

SFlow achieves context sensitivity by using a polymorphic type qualifier, poly, and viewpoint adaptation [4].

- **poly**: The poly qualifier expresses context sensitivity. poly is interpreted as tainted in some contexts and as safe in other contexts.

The subtyping hierarchy becomes

```
safe <: poly <: tainted
d```

The concrete value of poly is interpreted by the viewpoint adaptation operation. Viewpoint adaptation of a type $q'$ from the viewpoint of another type $q$, results in the adapted type $q''$. This is written as $q \triangleright q' = q''$. Viewpoint adaptation adapts fields, formal parameters, and method return values from the viewpoint of the context at the field access or method call. SFlow defines the viewpoint adaptation operation below:

```
- - tainted = tainted
- - safe = safe
q - poly = q
```

1 This is the desired subtyping. However, it is a well-known issue that subtyping is not always safe when mutable references are involved [14, 24, 28]. This is precisely the issue that necessitated inverse edges in CFL-Solver.
Again, there is analogy with CFL-Solver — constraint \( q_f \) corresponds to edge \( y \rightarrow x \) this and constraint \( q_f \) \( q_{\text{ret}} \) \( q_x \).

Let us return to the examples from Sect. 2. Consider

\begin{verbatim}
1 poly X id(poly X p) {
  2 return p;
  3 }
4 safe X b = id(a);
5 tainted X d = id(c);
\end{verbatim}

Statement return \( p \) enforces subtyping constraint \( p \). The call to \( \text{id} \) at 5 creates the following constraints (rule \( \text{TASSIGN} \) in Fig. 5):

\[
a <: q^5 \triangleright p \quad q^5 \triangleright \text{ret} \quad b
\]

It is easy to show that viewpoint adaptation preserves subtyping: that is, if \( p \) \( \text{ret} \) holds, then \( q^5 \triangleright p \). This constraint, combined with the above two constraints results in \( a \). This mirrors line 14 in CFL-Solver. Constraint \( a \triangleright q^5 \triangleright p \) corresponds to the \( \triangleright \) annotated edge from actual \( a \) to formal \( p \). \( p \) \( \triangleright \) corresponds the \( \triangleright \) annotated edge from \( p \) to \( \triangleright \) and finally, \( q^5 \triangleright \text{ret} \quad b \) corresponds to the \( \triangleright \) annotated edge from \( \triangleright \) to the left-hand-side of the call assignment \( b \).

SFlow is context-sensitive. \( \text{id} \) is polymorphic:

\[
\text{poly} \ X \ \text{id}(\text{poly} \ X \ p)
\]

At callsite 5, the poly type is instantiated to \( \text{safe} \) and at callsite 6, it is instantiated to \( \text{tainted} \). As discussed, SFlow enforces the following constraints at callsite 5:

\[
a <: q^5 \triangleright \text{poly} \quad q^5 \triangleright \text{poly} \quad \text{safe}
\]

\( q^5 \) \( \text{safe} \) satisfied the above constraints. SFlow enforces the following constraints at callsite 6:

\[
c <: q^6 \triangleright \text{poly} \quad q^6 \triangleright \text{poly} \quad \text{tainted}
\]

\( q^6 \) \( \text{tainted} \) satisfied the above constraints.

Consider the handling of structure-transmitted dependences in SFlow. Recall Fig. 3. It stipulates that fields are tainted or poly. This mirrors CFL-Solver. If the left-hand-side of a field read flows to sink, the field is forced to poly, “erasing” the field info and propagating \( \text{safe} \) towards writes of the field; otherwise, the field is tainted and right-hand-sides of writes to the field can remain tainted. Consider:

\begin{verbatim}
1 x.f = a;
2 x.g = c;
3 y = x;
4 b = y.f;
5 safe d = y.g;
\end{verbatim}

At field \( d = y.g \), SFlow enforces constraint \( y \triangleright g \quad \text{safe} \). Since field \( g \) is tainted or poly, but not \( \text{safe} \), \( g \) must be poly. As a result, constraint \( y \triangleright g \quad \text{safe} \) entails \( y \quad \text{safe} \) (because \( y \triangleright g \) poly equals \( y \)). At \( x.g = c \), SFlow creates constraint \( c <: x \triangleright g \), which again entails \( c <: x \). The assignment \( y = x \) enforces \( x \quad y \quad \text{safe} \). The three constraints

\[
c <: x \quad x <: y \quad y <: \text{safe}
\]
entail \( c <: \) safe, as expected. Note that \( x \) and \( y \) must be safe. However, field \( f \) as well as \( a \) and \( b \) may remain tainted. Again, SFlow mirrors CFL-Solver: the safe sink \( d \) is propagated backwards to \( c \), but \( a \) remains unaffected.

As a final example, consider the code in Fig. 2. Due to \( b = z.f \), SFlow enforces constraint \( z \triangleright f <: \) safe, which forces \( f \) to poly and entails constraint \( z <: \) safe. Since \( f \) is poly, statement \( f = p \) in set entails \( p <: \) this. The call to set at 12 deserves attention. SFlow creates the following constraints:

\[
\begin{align*}
  x <: q^{12} & \triangleright this \quad q^{12} \triangleright this <: x \quad a <: q^{12} \triangleright p
\end{align*}
\]

(Due to the mutation of this, SFlow enforces \( q^{12} \triangleright this <: x \). This is analogous to the inverse edge in CFL-Solver.) Constraints

\[
\begin{align*}
  a <: q^{12} \triangleright p & \quad p <: this \quad q^{12} \triangleright this <: x
\end{align*}
\]

entail \( a <: x \). Further, SFlow gives rise to these constraints

\[
\begin{align*}
  a <: x \quad x <: y & \quad y <: safe
\end{align*}
\]

mirroring the path from \( a \) to \( b \) in Fig. 2. The above constraints force \( a \) to be safe, as expected.

The above examples give intuition into SFlow and its connection to CFL-Solver. The theorems below formalize the connection. Lemma 3.1 states (roughly) that if CFL-Solver discovers a path from reference variable \( x \) to reference variable \( y \), then SFlow enforces appropriately that \( x \) is a subtype of \( y \). For example, if CFL-Solver discovers an \( N \)-path from \( x \) to \( y \), then SFlow enforces constraint \( x <: q \triangleright y \) for some value of \( q \).

**Lemma 3.1.** Let \( \text{CFL-Solver} \vdash c \) denote that CFL-Solver adds edge \( e \) to \( G \). Let \( \text{SFlow} \vdash c \) denote that SFlow enforces constraint \( c \). \( q, q' \in \{ \text{tainted}, \text{poly}, \text{safe} \} \).

1. CFL-Solver \( \vdash x \rightarrow y \Rightarrow \text{SFlow} \vdash x <: y 
2. CFL-Solver \( \vdash x \rightarrow y \Rightarrow \text{SFlow} \vdash x <: q \triangleright y 
3. CFL-Solver \( \vdash x \rightarrow y \Rightarrow \text{SFlow} \vdash q \triangleright x <: y 
4. CFL-Solver \( \vdash x \rightarrow y \Rightarrow \text{SFlow} \vdash q \triangleright x <: q' \triangleright y 

**Theorem 3.2.** Let \( \text{CFL-Solver} \vdash c \) denote that edge \( e \) is in \( G \) and let \( \text{SFlow} \vdash c \) denote that constraint \( c \) holds.

1. CFL-Solver \( \vdash x \rightarrow n \Rightarrow \text{SFlow} \vdash x <: safe 
2. CFL-Solver \( \vdash x \rightarrow n \Rightarrow \text{SFlow} \vdash x <: safe 
3. CFL-Solver \( \vdash x \rightarrow n \Rightarrow \text{SFlow} \vdash x <: poly 
4. CFL-Solver \( \vdash x \rightarrow n \Rightarrow \text{SFlow} \vdash x <: poly 

Theorem 3.2 follows directly from Lemma 3.1. For example, if CFL-Solver discovers an \( N \)-path from \( x \) to \( n \), then by Lemma 3.1 we have \( x <: q \triangleright n \). Since \( n \) is safe, \( x <: safe \).

4. A Type System Equivalent to CFL-Solver

SFlow is simple and therefore appealing. Unfortunately, it is stricter than CFL-Solver: it propagates sinks further and rejects programs that CFL-Solver deems safe. Theorem 3.2 reflects this. While a path from \( x \) to \( n \) in \( G \) implies that \( x \) is safe or poly in SFlow, \( x \) being safe or poly does not necessarily guarantee a path from \( x \) to \( n \).

This section elaborates on the imprecision in SFlow (Sect. 4.1) and proceeds to define a type system equivalent to CFL-Solver (Sect. 4.2). Finally, it outlines the type inference analysis (Sect. 4.3).

### 4.1 Imprecision in SFlow

The imprecision arises from rule (tcall). Essentially, (tcall) may force flow from an actual to the left-hand-side of the call assignment, even when there is no flow from the corresponding formal parameter to the return value. Suppose there is a method \( m \), such that there is no flow from its this to its ret. Suppose that \( m \)'s this flows to sink in context \( q' \), but it does not flow to sink in context \( q \). (More precisely, it would be this\( .f \) written in \( m \) that would flow to sink.) Also, let ret flow to sink in context \( q' \), but not in context \( q \).

Naturally, \( m \)'s this and ret should both be poly. Thus this \( <: \) ret, which in SFlow leads to

\[
y <: q' \triangleright this \quad <: \quad q' \triangleright ret \quad <: x
\]

where \( y \) and \( x \) are the receiver and left-hand-side, respectively, at callsite \( j \). Since \( x \) flows to sink, \( x \) is safe, and the above constraints force \( y \) to be safe, unnecessarily.

The example in Fig. 6 illustrates this issue and differentiates with CFL-Solver. Reference variables \( a \) and \( a1 \) refer to two distinct A objects. Field \( a.f \) flows to sink, but \( x1 \), returned out of \( a.m() \), does not flow to sink (lines 7-9). In contrast, \( a1.f \) does not flow to sink, but \( a1.m() \) does (lines 11-12). Fig. 6 shows the edges added due to program statements by CFL-Solver. (CFL-Solver proceeds to add path edges accordingly. These edges are not shown.) Note that \( a1 \) and \( x1 \) remain unaffected by the sink because there is no C-path from \( a1 \) or \( x1 \) to a sink. \( x2 \) is not in the graph.

SFlow enforces the following constraints at line 7:

\[
a <: q^7 \triangleright this \quad q^7 \triangleright this <: a \quad q^7 \triangleright ret \quad <: x1 \quad x1 <: q^7 \triangleright ret
\]

Due to the mutations to this and to \( x1 \), there are constraints in the expected direction and in the opposite direction. Due
to line 9, a is safe and thus this \( < : \) poly. SFow creates the following constraints at line 11:

\[
\text{a1} < : q^{11} > \text{this} \quad q^{11} > \text{this} < : \text{a1} \quad q^{11} > \text{ret} < : \text{sink1}
\]

The last constraint forces \( \text{ret} < : \) poly.

If this is safe, then due to \( \text{a1} < : q^{11} > \text{this}, \text{a1} \) is forced to be safe, unnecessarily; if \( \text{ret} \) is safe, then due to \( x1 < : q^{7} > \text{ret}, \) \( x1 \) is forced to be safe, again unnecessarily.

As we argued earlier, this being poly is the natural choice: this, \( f \) is safe in context \( q^{7} \) and unaffected in \( q^{11} \). Similarly, \( \text{ret} \) being poly is the natural choice: ret is safe in context \( q^{11} \) and unaffected in \( q^{7} \). However, then both \( q^{7} \) and \( q^{11} \) would be safe, and safe would propagate to both \( \text{a1} \) and \( x1 \).

The problem is that in SFow, callsite contexts \( q^{i} \) play two roles: (1) they propagate formal-to-return-value dependences to give rise to actual-to-left-hand-side of call assignment dependences, akin to line 14 in CFL-Solver, and (2) they account for \( N \)-paths and \( P \)-paths, akin to lines 15-18 in CFL-Solver. In most cases, if both the parameter and the return value are poly then there is flow from the parameter to the return value (e.g., \( p < : \text{ret in id} \)), and SFow is precise. However, when there is no flow from the parameter to the return value, SFow is imprecise as in Fig. 6.

### 4.2 Type System

We now present SFowCFL, a type system that is equivalent to CFL-Solver and thus strictly more precise than SFow.

The typing rules for SFowCFL appear in Fig. 7. SFowCFL augments SFow with a set of local constraints \( L \) and a set of rules for the local constraints in \( L \). Constraints in \( L \) are of form \( x \rightarrow \rightarrow y \), \( y.f \rightarrow \rightarrow x \) or \( x \rightarrow \rightarrow y.f \), where \( x \) and \( y \) are local variables in the same method \( m \). \( L \vdash x \rightarrow \rightarrow y \) means that there is flow from \( x \) to \( y \). All constraints in \( L \) must obey the subtyping relation, e.g., if \( x \rightarrow \rightarrow y \) is in \( L \), then \( x < : y \) must hold in \( \Gamma \). This is ensured by rule (well-formed \( L \) in \( \Gamma \)).

Rule (trans) adds constraints \( x \rightarrow \rightarrow y \) to \( L \) due to transitivity. Rule (trans) discovers dependences from formal parameters to return values. We call these dependences method summary constraints. Rules (linearize-tread) and (linearize-twrite) add constraints \( x \rightarrow \rightarrow y \) when the adapted field is poly (if \( f \) is poly, then \( x.f \rightarrow \rightarrow y \) entails \( x \rightarrow \rightarrow y \) because \( x > : f \) evaluates to \( x \)). These rules “erase” field information, akin to the way CFL-Solver “erases” field annotations \( [t] \text{ and } [r] \) when \( f \) flows into a sink. Just as with CFL-Solver, erasure happens only if \( f \) flows into a sink. Intuitively, constraints \( x \rightarrow \rightarrow y \) in \( L \) correspond to d-paths in \( G \).

SFowCFL improves over SFow because its (tcall) separates role (1), the propagation of formal-to-return-value dependences (e.g., \( p \rightarrow \rightarrow \text{ret} \)), which must give rise to actual-to-left-hand-side dependences (\( act(p) \rightarrow \rightarrow act(\text{ret}) \)), from role (2), the handling of \( N \)-paths and \( P \)-paths.

Rule (tcall) has two parts. The first part propagates all method summary constraints, i.e., formal-to-return-value dependences. For example, in method id, there is method summary constraint \( p \rightarrow \rightarrow \text{ret} \). At call \( x = \text{id}(y) \) (tcall) requires that \( L \) entails \( y \rightarrow \rightarrow x \), as expected. The second part creates constraints that link actuals to formals, such as \( q_{y} < : q^{7}_{p} > q_{\text{this}} \). These constraints capture \( N \)-paths and \( P \)-paths. Qualifiers \( q^{7}_{y} \) are distinct, which avoids unnecessary propagation from actuals to left-hand-sides of call assignments.

Let us return to the example in Fig. 6. SFowCFL enforces the following constraints at line 7:

\[
\text{a} < : q^{7}_{1} > \text{this} \quad q^{7}_{1} > \text{this} < : \text{a} \quad q^{7}_{2} > \text{ret} < : \text{x1} \quad \text{x1} < : q^{7}_{3} > \text{ret}
\]

SFowCFL enforces the following constraints at line 11:

\[
\text{a1} < : q^{11}_{1} > \text{this} \quad q^{11}_{1} > \text{this} < : \text{a1} \quad q^{11}_{3} > \text{ret} < : \text{sink1}
\]
```
1: procedure SFLOWCFL-INFER
2:   for each variable x do S(x) ← {tainted, poly, safe} end for
3:   for each q_i at call site i do S(q_i) ← {tainted, poly, safe} end for
4:   for each field f do S(f) ← {tainted, poly} end for
5:   for each statement s do Add constraints for s to C end for
6:   repeat
7:     for each c in C do
8:       SOLVECONSTRAINT(c)
9:         if c is x < y then f and S(f) is {poly} then ▷ Case 1
10:             Add x < y into C
11:         else if c is x < y and S(f) is {poly} then ▷ Case 2
12:             Add x < y into C
13:         else if c is x < y then ▷ Case 3
14:             for each y : z in C do Add x < z to C end for
15:         for each w : x in C do Add w < y to C end for
16:         for each w : q_i > x and q_i > y : z in C do ▷ Case 4
17:             Add w < z to C
18:         end for
19:     end if
20: end for
21: until S remains unchanged
22: end procedure
```

Recall that a and sink1 are safe, this and ret can be poly, q_i = safe, q_i = tainted, q_j = tainted and q_k = safe satisfy the above constraints. Thus, a1 and x1 can be tainted, i.e., unaffected by the safe sink, precisely as in CFL-Solver.

### 4.3 Type Inference

Given sources and sinks, type inference derives a valid typing, i.e., an assignment from program variables to type qualifiers that type checks with the typing rules in Fig. 7.

SFlow and SFlowCFL permit many valid typings. For example, if there are no tainted sources, one can type all variables safe and all fields poly and this assignment type checks. This typing is hardly useful. Intuitively, we would like to infer a typing that minimizes the impact of safe sinks through the code, just as CFL-Solver minimizes this impact.

Our type inference algorithm computes the typing that minimizes the impact of safe. It initializes all variables to sets of qualifiers. S is the mapping from variables to sets of qualifiers. Programmer-annotated variables, including sources and sinks, are initialized to the singleton set that contains the provided type qualifier. For example, sources and sinks from the annotated library map to \{tainted\} and \{safe\}, respectively. Fields f are initialized to \(S(f) = \{tainted, poly\}\). All other variables and all \(q_i\) are initialized to the maximal set of qualifiers \{tainted, poly, safe\}.

The algorithm proceeds to create constraints for all program statements according to the typing rules in Fig. 7. It adds those constraints to set of constraints \(C\). For example, at \(x.f = y\) the algorithm adds \(y : x \triangleright f\) to \(C\). If \(y\) is not already, it adds \(x \triangleright f < y\) as well. Then the algorithm iterates over constraints \(c \in C\) and calls SOLVECONSTRAINT(c).

Figure 8. Type inference for SFlowCFL. Lines 2-5 initialize \(S\) and \(C\). Cases 1 and 2 add \(x < y\) into \(C\) because \(y \triangleright poly\) always yields \(y\). These cases mirror (LINEARIZE-TREAD) and (LINEARIZE-TWRITE). Case 3 adds constraints due to transitivity. It mirrors (TRANS). Case 4 adds constraints between actual(s) and left-hand-side(s) at calls: if there are constraints \(w <: q_i \triangleright x\) (flow from actual \(w\) to formal \(x\)) and \(q_i > y : z\) (flow from return \(y\) to left-hand-side \(z\)), and also \(x < y\) (flow from formal to return value), Case 4 adds \(w <: z\). Note that line 4 calls SOLVECONSTRAINT(c): the solver infers new constraints, which remove additional infeasible qualifiers from \(S\). This process repeats until \(S\) remains unchanged.

The following theorems relate CFL-Solver to \(T\). Theorem 4.1 states that when \(T\) exists, \(x\) is inferred as tainted if and only if CFL-Solver discovers no paths from \(x\) to sink \(n\).

**Theorem 4.1.** Let \(T\) exist.

1. CFL-Solver \(\vdash x \overset{4}{\rightarrow} n \iff T(x) : <\text{safe}\)
2. CFL-Solver \(\vdash x \overset{N}{\rightarrow} n \iff T(x) : <\text{safe}\)
3. CFL-Solver \(\vdash x \overset{C}{\rightarrow} n \iff T(x) : <\text{poly}\)

**Theorem 4.2.** CFL-Solver \(\vdash x \overset{C}{\rightarrow} n\), where \(x\) is tainted source \(\iff T = \text{error}\).

\(^2T\) subsamples every valid typing. Thus, it can be viewed as a principal type.
It is worth noting that $T$, computed by SFlowCFL-Infer, practically always type checks with SFlow [14, 15]. SFlow is simpler and therefore more appealing than SFlowCFL. The results in this paper mean that even in the rare case when $T$ does not type check with SFlow, there is no flow from sources to sinks.

5. Inheritance and Virtual Calls

The standard approach to handling of virtual calls in CFL-Solver would be to compute a set of actual target methods at each virtual call $i$, and then run lines 9-11 (Fig. 4) for each target method. One can compute this set by using established techniques such as points-to analysis [17, 22] or CHA [3]. Unfortunately, such handling precludes modularity and compositionality. Suppose we analyzed an open library $P$ using CHA. If we composed $P$ with user code that extended one or more of $P$’s classes, we would need to reanalyze $P$ because an overriding method could have introduced new flows through callbacks, and changed the analysis result.

We handle inheritance and virtual calls with standard function subtyping augmented with reference immutability. Let method $m$, where $\text{typeof}(m) = q_{\text{this}}, q_{\text{pm}} \rightarrow q_{\text{ret}}$, override $m$, where $\text{typeof}(m) = q_{\text{this}}, q_{\text{pm}} \rightarrow q_{\text{ret}}$. SFlow requires

$$
q_{\text{this}} <: q_{\text{this}}, \text{RetIm}(\text{this}_n) \neq \text{readonly} \Rightarrow q_{\text{this}} <: q_{\text{this}} \\
q_{\text{pm}} <: q_{\text{pm}}, \text{RetIm}(\text{pm}_n) \neq \text{readonly} \Rightarrow q_{\text{pm}} <: q_{\text{pm}} \\
q_{\text{ret}} <: q_{\text{ret}}, \text{RetIm}(\text{ret}_m) \neq \text{readonly} \Rightarrow q_{\text{ret}} <: q_{\text{ret}}
$$

The above constraints guarantee that if there is a pair of parameters of overriding method $n$ such that $p_n <: p'_n$, then the same subtyping holds for the corresponding pair of parameters of overridden method $m$: $p_m <: p'_m$. (Recall that $p'$ is usually ret.) For example, suppose that $n$ has statement return $p$ but $m$ does not. This entails $p_n <: \text{ret}_n$. Due to the above constraints, we have $p_m <: p_m$ (contravariant arguments) and $\text{ret}_m <: \text{ret}_m$ (covariant return values), which yields $p_m <: \text{ret}_m$. Thus, return $p$ in overriding method $n$ will give rise to the expected actual-to-left-hand-side dependence at calls to overridden method $m$. This handling is modular, in the sense that we can analyze any given set of classes $P$. We can compose user code with $P$ without reanalysis of $P$ — we simply must check that the above constraints hold for every $n$ and $m$, where $n$ is a user method that overrides $m$ from $P$.

SFlowCFL imposes the following requirement, in addition to the above subtyping constraints:

$$L \vdash p_n \rightarrow p'_n \Rightarrow L \vdash p_m \rightarrow p'_m$$

SFlowCFL requires explicit method summary constraints and the above implication ensures that calls to $m$ give rise to actual-to-left-hand-side dependences triggered by $n$. The subtyping constraints capture $N$-paths and $P$-paths.

Finally, CFL-Solver calls $\text{BiEdge}(\text{this}_n, \text{this}_m, \text{this}_n, d, d)$, $\text{BiEdge}(p_n, p_m, \text{pm}_n, \text{pm}_m, d, d)$ and $\text{BiEdge}(\text{ret}_m, \text{ret}_n, \text{ret}_m, d, d)$. Thus, CFL-Solver creates edges $\text{this}_m \rightarrow \text{this}_n$, from $p_m \rightarrow p_n$ and from $\text{ret}_m \rightarrow \text{ret}_n$ plus the corresponding inverse edges when necessary.

6. Application to Taint Analysis for Android

One compelling application of SFlow, SFlowCFL and SFlowCFL-Infer is static taint analysis for Android. Taint analysis for Android detects leaks of sensitive data (e.g., phone data, location data) to untrusted places (e.g., the Internet, log files). Static taint analysis for Android has received considerable attention during the last 3 years [2, 7, 9, 11, 16, 18, 19, 35]. One notable effort is DARPA’s Automated Program Analysis for Cybersecurity (APAC) program, which has been active for 2 years. Yet a solution remains elusive. The state-of-the-art is FlowDroid, a highly-precise context-, flow-, field-, object-sensitive and lifecycle-aware analysis for Android [2]. Unfortunately, FlowDroid is heavyweight and memory-intensive. Furthermore, it does not detect leaks in apps from the Google Play store.

Meanwhile, we worked on a type inference and checking framework, which we instantiated with several type systems and their corresponding inferences [12–14]. SFlow and SFlowCFL-Infer were built as an instance of the framework and incidentally, SFlowCFL-Infer came to be an effective taint analysis for Android. SFlowCFL-Infer outperforms FlowDroid. It scales well and detects numerous leaks in popular Android apps from the Google Play Store [15]. SFlowCFL-Infer’s success prompted our investigation into its connection to CFL-reachability and the discovery of CFL-Solver and SFlowCFL.

The connection to CFL-reachability is important because it can help address error reporting, a known issue with type-based approaches: given a type error, what source-sink path is to blame for the type error? Our current approach is effective but requires some manual effort and has false positives [15]. With CFL-reachability, we envision 0-false-positive error reports, where each type error comes with source-sink paths.

In conclusion to this section, consider the FieldSensitivity2 example in Fig. 9. This example is refactored from DroidBench [9], a set of micro benchmarks for taint analysis for Android; it is representative of real-world leaks.

The return of TelephoneManager.getSimSerialNumber (line 10) is a source and the parameter msg of SmsManager.sendTextMessage (line 16) is a sink. The serial number of the SIM card is obtained and stored into a Data object. Later, it is retrieved from the Data object and sent out through an SMS message without user consent. Fig. 9 demonstrates SFlowCFL-Infer (Fig. 8).

7. Related Work

Shankar et al. present an integrity type system for detecting string format vulnerabilities in C programs [30]. The type system has two type qualifiers, tainted and untainted; polymorphism is not part of the core system. In contrast, SFlow and SFlowCFL-Infer handle polymorphism naturally. Ernst et al. present IFC, an integrity type system for detecting privacy leaks in Android apps [7]. IFC is similar to SFlow but it requires annotations and works only on Java source. In contrast
to previous work on integrity types, this paper focuses on the connection of integrity types to CFL-reachability.

Rehof and Fahndrich [25] study the connection between type-based flow analysis with polymorphic subtyping and CFL-reachability. They do not allow for polymorphism in the type structure, which essentially entails graphs without brackets. In contrast, we handle graphs with both parentheses and brackets. Furthermore, Rehof and Fahndrich [25] do not discuss mutable references. Fahndrich et al. [8] present an application of [25] to context-sensitive Steensgard-style points-to analysis for C. This work handles mutable references, because it uses equality (i.e., unification) constraints. Equality constraints is the standard approach to the handling of mutable references [10, 28, 30]. However, this approach is imprecise [21]. A key novelty in our work is the use of reference immutability to allow for limited subtyping.

Sridharan and Bodik [31] present refinement-based points-to analysis for Java using CFL-reachability. Recently, Shang et al. [29] and Lu et al. [20] build upon this work towards an incremental demand-driven points-to analysis for Java. Xu et al. [34] improve the scalability of CFL-reachability-based points-to analysis. These works focus on points-to analysis and all require heap abstraction. Heap abstraction requires handling of reflective object creation, which is inherently difficult; also, it may preclude modularity and compositionality. One key achievement of our work is the CFL-reachability-based integrity analysis, which avoids heap abstraction and is modular and compositional. Another key difference with previous work is our focus on type-based approaches and their connection to CFL-reachability.

There is a lot of recent work on static taint analysis for Android [9, 11, 16, 19, 35]. These analyses typically use points-to and dataflow-based approaches. Type-based taint analysis appears to be better suited to the problem. Volpano et al. [33] and Myers [23] present type systems for secure information flow. These systems are substantially more complex and powerful than SFlow. They focus on type checking and do not include type inference, or include only local type inference.

8. Conclusion

We presented SFlow, a context-sensitive integrity type system and we give an interpretation of SFlow in terms of CFL-reachability. We proposed SFlowCFL, a more precise integrity type system, and SFlowCFL-Infer, the corresponding type inference analysis, which is equivalent to CFL-reachability. SFlowCFL-Infer is an effective taint analysis for Android detecting numerous privacy leaks in Android apps from the Google Play Store.
References


