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:

$$\mathsf{neg} <: \mathsf{poly} <: \mathsf{pos}$$

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* x to a "negative" *sink* y. 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 y (mutable is the "negative" qualifier, e.g., y is mutable in y.f = z). 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.[1] 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

[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, 26]. 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-rechability-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).

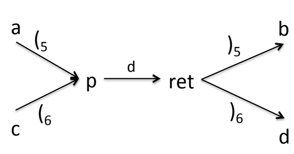
2.1 Overview of CFL-reachability

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:

```
1  X id(X p) {
2    return p;
3  }

5  X b = id(a);
6  X d = id(c);
```



Intraprocedural (direct) flows are represented with d-annotated edges. In the above example, the edge from p to ret represents that parameter p flows to return value ret inside id. For the remainder of the paper, we treat return statements as assignments to a special variable ret (e.g., return p is ret = p). Flow from an actual argument to a formal parameter is represented with an $(_i$-annotated edge, where i is the callsite index. The edge from a to p represents the fact that at callsite 5, a flows to p. Finally, flow from the return value to the left-hand-side of the call assignment is represented with $)_i$-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 $(_5\ \mathtt{d}\ )_5$ belongs to the language. However, a does not flow to d because $(_5\ \mathtt{d}\ )_6$ 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).

$$
\begin{aligned}
C &::= P\ N \mid P \mid N\\
P &::= M \mid)_i \mid M\ P \mid)_i\ P\\
N &::= M \mid (_i \mid M\ N \mid (_i\ N\\
M &::= \mathsf{d} \mid (_i\ M\ )_i \mid M\ M
\end{aligned}
$$

(a) Context-free grammar C

$$
\begin{aligned}
F &::= P\ N \mid P \mid N\\
P &::= M \mid]_\mathsf{f} \mid M\ P \mid]_\mathsf{f}\ P\\
N &::= M \mid [_\mathsf{f} \mid M\ N \mid [_\mathsf{f}\ N\\
M &::= \mathsf{d} \mid [_\mathsf{f}\ M\ ]_\mathsf{f} \mid M\ M
\end{aligned}
$$

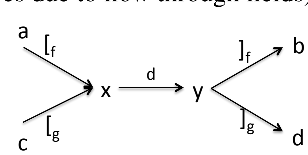
(b) Context-free grammar F

Figure 1. (a) C 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.

Fig. 1(a) shows grammar C, which handles call-transmitted dependences. M-paths are paths of balanced (matched) parentheses that represent intraprocedural flows. In the id example, d and $(_5\ \mathtt{d}\ )_5$ are M-paths. N-paths represent paths with outstanding calls, e.g., the path from a to ret has string $(_5\ \mathtt{d}$, which is generated by N. P-paths represent paths due to returns, e.g., the path from ret to b has string $)_5$ generated by P. We abuse notation slightly by having C denote both the grammar and the starting nonterminal.

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

```
1  x.f = a;
2  x.g = c;
3  y = x;
4  b = y.f;
5  d = y.g;
```

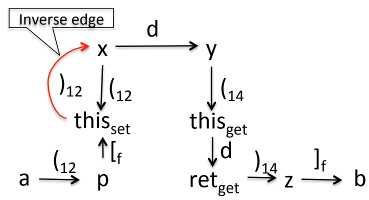


Above, $[_\mathsf{f}$-annotated edges model field writes and $]_\mathsf{f}$-annotated edges model field reads. The edge from a to x represents the fact that a flows to field f of x (due to x.f = a at line 1). Structure-transmitted dependences can be modeled using an analogous context-free grammar, shown in Fig. 1(b). Here a flows to b because string $[_\mathsf{f}\ \mathtt{d}\ ]_\mathsf{f}$ belongs to the language generated by the grammar in Fig. 1(b). However, a does not flow to d because $[_\mathsf{f}\ \mathtt{d}\ ]_\mathsf{g}$ does not belong to the language.

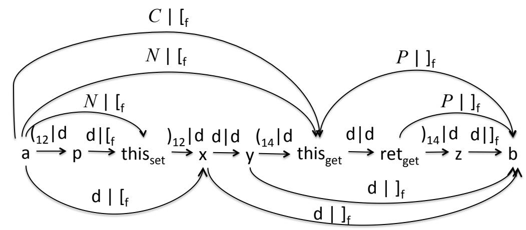
```
1  class X {
2    Y f;
3    void set(Y p) {
4      this.f = p;
5    }
6    X get() {
7      return this;
8    }
9  }
```

(a) Sample code

```
10 main() { ...
11   x = new X()
12   x.set(a);
13   y = x
14   z = y.get();
15   safe Y b = z.f
16   // b is a sink
17 }
```



(b) Edges due to statements (added by lines 4-12 in Fig. 4).



(c) Inferred paths (lines 13-19 in Fig. 4). Not all paths are shown.

Figure 2. An example with mutable references and interleaving of parentheses and brackets.

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:

```
X get_f() {
  return this.f;
}

x.f = a;
b = x.get_f()
```

$$\mathsf{a} \xrightarrow{[_f} \mathsf{x} \xrightarrow{(_6} \mathsf{this} \xrightarrow{]_f} \mathsf{ret} \xrightarrow{)_6} \mathsf{b}$$

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 right-hand-side y 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 $\xrightarrow{(_{12}}$ $\mathsf{this_{set}}$ 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 $\mathsf{this_{set}}$ is *mutated*. The value written into field f of $\mathsf{this_{set}}$ 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 $\mathsf{this_{set}}$: x $\xrightarrow{(_{12}}$ $\mathsf{this_{set}}$ and inverse edge $\mathsf{this_{set}}$ $\xrightarrow{)_{12}}$ x. Note that the annotation is reversed as well, $(_{12}$ becomes $)_{12}$; this expresses the fact that the value written into f of $\mathsf{this_{set}}$ is *returned* to the caller of set. The inverse edge is added because there is mutation of $\mathsf{this_{set}}$ at this.f = p. With the inverse edge, there is a path from a to b with string $(_{12}\ [_f\ )_{12}\ \mathsf{d}\ (_{14}\ \mathsf{d}\ )_{14}\ ]_f$ 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 → ... → 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 → 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.

cd	::=	class C extends D $\{\overline{fd}\ \overline{md}\}$	*class*
fd	::=	t' f	*field*
md	::=	t m(t this, t x) $\{\ \overline{t\ \mathsf{y}}$ s; return y $\}$	*method*
s	::=	s; s | x = new t() | x = y | x = y.f | y.f = x | x = y.m(z)	*statement*
t	::=	q C	*qualified type*
t'	::=	q' C	*qualified field type*
q	::=	tainted | poly | safe	*qualifier*
q'	::=	tainted | poly	*field qualifier*

Figure 3. Syntax. C and D are class names, f is a field name, m is a method name, and x, y, and z are names of local variables, formal parameters, or parameter this. As in some of the code examples, this is explicit. For simplicity, we assume all names are unique.

1: **procedure** CFL-SOLVER
2: Initialize G with sink node n
3: **repeat**
4: **for** each (ASSIGN) $x = y$ **do** BIEDGE(x,y,x,d,d) **end for**
5: **for** each (WRITE) $y.f = x$ **do** BIEDGE(y.f,x,y,d,$[_f$) **end for**
6: **for** each (READ) $x = y.f$ **do** BIEDGE(x,y,x,d,$]_f$) **end for**
7: **for** each (CALL) $i : x = y.m(z)$ **do**
8: let m : this, p $\rightarrow$ ret be the static target at i
9: BIEDGE(this at i,y,this,$(_i$,d)
10: BIEDGE(p at i,z,p,$(_i$,d)
11: BIEDGE(x, ret, x, $)_i$, d)
12: **end for**
13: **for** each $x \xrightarrow{d \mid f} y \xrightarrow{d \mid f'} z \in G$ **do** EDGE(x,z,d,$f \oplus f'$) **end for**
14: **for** each $x \xrightarrow{(_i \mid d} p \xrightarrow{d \mid f} p' \xrightarrow{)_i \mid d} z \in G$ **do** EDGE(x,z,d,f) **end for**
15: **for** each $x \xrightarrow{Y \mid f} y \in G$ s.t. $X ::= Y$ **do** EDGE(x,y,X,f) **end for**
16: **for** each $x \xrightarrow{Y \mid f} y \xrightarrow{Z \mid f'} z \in G$ s.t. $X ::= YZ$ **do**
17: EDGE(x,z,X,$f \oplus f'$)
18: **end for**
19: **until** G remains unchanged
20: **end procedure**
21: **procedure** BIEDGE(lhs,s,t,c,f)
22: EDGE(s,t,c,f)
23: **if** $ReIm(lhs) \neq$ readonly **then** EDGE(t,s,$\overline{c}$,$\overline{f}$) **end if**
24: **end procedure**
25: **procedure** EDGE(s,t,c,f)
26: **if** $t \xrightarrow{C \mid f'} n \in G \ \wedge \ f \in L(F) \ \wedge \ f$ is $[_f \Rightarrow]_f \in G$ **then**
27: Add $s \xrightarrow{c \mid f} t$ to G
28: **end if**
29: **end procedure**

Figure 4. CFL-reachability-based integrity analysis. $x \xrightarrow{c \mid f} y \in G$ tests if the edge is in G, and $]_f \in G$ tests if there is an edge in G with field annotation $]_f$.

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 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 $x \xrightarrow{c \mid f} y$. 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 readonly, this guarantees that x is never used to mutate the referenced object nor anything it references. If x is readonly, all of the following are forbidden:

- x.f = z
- x.setField(z) where setField sets a field of its receiver
- y = id(x); y.f = z
- x.f.g = z
- y = x.f; y.g = z

Function $ReIm(expr)$ returns the ReIm type of the expression argument. Procedure 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 and t — the source and target of the edge respectively, and c and f — the call and field annotations of the new edges. CFL-Solver calls BIEDGE at every program statement (lines 4-12). If the left-hand-side of the assignment is readonly according to ReIm, BIEDGE adds only one edge, from s to t. Otherwise, BIEDGE adds an inverse edge from t to s as well. $\overline{c}$ and $\overline{f}$ are as expected: $\overline{[_f} =]_f$, $\overline{]_f} = [_f$, $\overline{(_i} =)_i$, $\overline{(_i} =)_i$, $\overline{d} = d$. As an example, consider (WRITE) y.f = x. If left-hand-side y.f is readonly, then BIEDGE creates edge $x \xrightarrow{d \mid [_f} y$. Otherwise, it creates edges $x \xrightarrow{d \mid [_f} y$ and $y \xrightarrow{d \mid]_f} x$. 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 $x \xrightarrow{d \mid f} y$ and edge $y \xrightarrow{d \mid f'} z$ and adds a new edge $x \xrightarrow{d \mid f \oplus f'} z$, 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 $y \xrightarrow{(_{14} \mid d} this_{get} \xrightarrow{d \mid d} ret_{get}$. Line 15 adds edge $this_{get} \xrightarrow{N \mid d} ret_{get}$ due to production $N ::= M$. Lines 16-18 then add edge $y \xrightarrow{N \mid d} ret_{get}$ due to production $N ::= (_i\ N$, recording the N-path from y to ret_{get}.[2] 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 com-

[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 $[_{\mathsf{f}}$, the corresponding field read $]_{\mathsf{f}}$ must be in G. The role 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 $]_{\mathsf{f}}$ is never added to G and the sink does not propagate towards writes of f. Consider

```
x.f = a;
x.g = c;
y = x;
b = y.f;
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:

$$\mathsf{safe} <: \mathsf{tainted}^{3}$$

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:

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

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

```
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

$$\mathsf{safe} <: \mathsf{poly} <: \mathsf{tainted}$$

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 $q \rhd q' = q''$. 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:

$$\begin{array}{ccccc} \_ & \rhd & \mathsf{tainted} & = & \mathsf{tainted} \\ \_ & \rhd & \mathsf{safe} & = & \mathsf{safe} \\ q & \rhd & \mathsf{poly} & = & q \end{array}$$

[3] This is the desired subtying. 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.

$$\frac{\substack{\text{(TASSIGN)}\\ \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{x} \\ \mathit{ReIm}(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_\mathsf{x} <: q_\mathsf{y}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

$$\frac{\substack{\text{(TWRITE)}\\ \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad q_\mathsf{x} <: q_\mathsf{y} \rhd q_\mathsf{f} \\ \mathit{ReIm}(\mathsf{y.f}) \neq \mathsf{readonly} \Rightarrow q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

$$\frac{\substack{\text{(TREAD)}\\ \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x} \\ \mathit{ReIm}(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_\mathsf{x} <: q_\mathsf{y} \rhd q_\mathsf{f}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

$$\frac{\substack{\text{(TCALL)}\\ \mathit{typeof}(\mathsf{m}) = q_\mathsf{this}, q_\mathsf{p} \to q_\mathsf{ret} \\ \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{z}) = q_\mathsf{z} \quad \Gamma(i) = q^i \\ q_\mathsf{y} <: q^i \rhd q_\mathsf{this} \quad \mathit{ReIm}(\mathsf{this\ at}\ i) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_\mathsf{this} <: q_\mathsf{y} \\ q_\mathsf{z} <: q^i \rhd q_\mathsf{p} \quad \mathit{ReIm}(\mathsf{p\ at}\ i) \neq \mathsf{readonly} \Rightarrow q^i \rhd q_\mathsf{p} <: q_\mathsf{z} \\ q^i \rhd q_\mathsf{ret} <: q_\mathsf{x} \quad \mathit{ReIm}(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_\mathsf{x} <: q^i \rhd q_\mathsf{ret}}}{\Gamma \vdash i\colon \mathsf{x} = \mathsf{y.m(z)}}$$

Figure 5. SFlow type system. Function *typeof* retrieves the SFlow types of fields and methods, Γ is a type environment that maps variables to SFlow qualifiers.

The underscore denotes a "don't care" value. Qualifiers tainted and safe do not depend on the viewpoint (context). Qualifier poly depends on the viewpoint: e.g., if the viewpoint (context) is tainted, then poly is interpreted as tainted. Viewpoint adaptation is applied at field accesses and method calls; we elaborate upon this in the next section.

3.3 Type System

The SFlow typing rules are defined in Fig. 5. The key to the relation of CFL-Solver to SFlow is that (roughly speaking) if CFL-Solver finds a path from x to y, SFlow guarantees x $<:$ y. (For the remainder of the paper, we often write x instead of q_x for the type qualifier of x.)

Rule (TASSIGN) enforces the expected subtyping constraints. If the left-hand-side of the assignment is not readonly according to ReIm, (TASSIGN) creates an additional subtyping constraint in the opposite direction, just as CFL-Solver creates an inverse edge. The rules for field access, (TREAD) and (TWRITE), adapt field f from the viewpoint of *receiver* y and then enforce the subtyping constraints.

The rule for method call, (TCALL), adapts formal parameters this and p and return value ret from the viewpoint of *callsite context* q^i, and creates the subtying constraints that capture flows from actual arguments to formal parameters, and from return value to the left-hand-side of the call assignment. Again, there is analogy with CFL-Solver — constraint $q_\mathsf{y} <: q^i \rhd q_\mathsf{this}$ corresponds to edge $\mathsf{y} \xrightarrow{(_i} \mathsf{this}$ and constraint $q^i \rhd q_\mathsf{ret} <: q_\mathsf{x}$ corresponds to edge $\mathsf{ret} \xrightarrow{)_i} \mathsf{x}$.

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

```
1  poly X id(poly X p) {
2    return p;
3  }

5  safe X b = id(a);
6  tainted X d = id(c);
```

Statement return p enforces subtyping constraint p $<:$ ret. The call to id at 5 creates the following constraints (rule (TCALL) in Fig. 5):

$$\mathsf{a} <: q^5 \rhd \mathsf{p} \quad q^5 \rhd \mathsf{ret} <: \mathsf{b}$$

It is easy to show that viewpoint adaptation preserves subtyping: that is, if p $<:$ ret holds, then $q^5 \rhd \mathsf{p} <: q^5 \rhd \mathsf{ret}$ holds. This constraint, combined with the above two constraints results in a $<:$ b. This mirrors line 14 in CFL-Solver. Constraint $\mathsf{a} <: q^5 \rhd \mathsf{p}$ corresponds to the $(_5$-annotated edge from actual a to formal p, p $<:$ ret corresponds the d-annotated edge from p to ret, and finally, $q^5 \rhd \mathsf{ret} <: \mathsf{b}$ corresponds to the $)_5$-annotated edge from ret to the left-hand-side of the call assignment b.

SFlow is context-sensitive. id is polymorphic:

poly X id(**poly** X p)

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

$$\mathsf{a} <: q^5 \rhd \mathsf{poly} \quad q^5 \rhd \mathsf{poly} <: \mathsf{safe}$$

$q^5 = \mathsf{safe}$ satisfied the above constraints. SFlow enforces the following constraints at callsite 6:

$$\mathsf{c} <: q^6 \rhd \mathsf{poly} \quad q^6 \rhd \mathsf{poly} <: \mathsf{tainted}$$

$q^6 = \mathsf{tainted}$ 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 safe towards writes of the field; otherwise, the field is tainted and right-hand-sides of writes to the field can remain tainted. Consider:

```
x.f = a;
x.g = c;
y = x;
b = y.f;
safe d = y.g;
```

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

$$\mathsf{c} <: \mathsf{x} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} <: \mathsf{safe}$$

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 this.f = p in set entails p $<:$ this. The call to set at 12 deserves attention. SFlow creates the following constraints:

$$\mathsf{x} <: q^{12} \triangleright \mathsf{this} \quad q^{12} \triangleright \mathsf{this} <: \mathsf{x} \quad \mathsf{a} <: q^{12} \triangleright \mathsf{p}$$

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

$$\mathsf{a} <: q^{12} \triangleright \mathsf{p} \quad \mathsf{p} <: \mathsf{this} \quad q^{12} \triangleright \mathsf{this} <: \mathsf{x}$$

entail a $<:$ x. Further, SFlow gives rise to these constraints

$$\mathsf{a} <: \mathsf{x} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} <: \mathsf{safe}$$

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-Sovler. 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* CFL-SOLVER $\vdash e$ *denote that* CFL-SOLVER *adds edge* e *to* G. *Let* SFLOW $\vdash c$ *denote that* SFLOW *enforces constraint* c. $q, q' \in \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

1. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{d} \mathsf{y} \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: \mathsf{y}$
2. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{N} \mathsf{y} \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: q \triangleright \mathsf{y}$
3. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{P} \mathsf{y} \;\Rightarrow\;$ SFLOW $\vdash q \triangleright \mathsf{x} <: \mathsf{y}$
4. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{PN} \mathsf{y} \;\Rightarrow\;$ SFLOW $\vdash q \triangleright \mathsf{x} <: q' \triangleright \mathsf{y}$

THEOREM 3.2. *Let* CFL-SOLVER $\vdash e$ *denote that edge* e *is in* G *and let* SFLOW $\vdash c$ *denote that constraint* c *holds.*

1. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{d} n \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: \mathsf{safe}$
2. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{N} n \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: \mathsf{safe}$
3. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{P} n \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: \mathsf{poly}$
4. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{PN} n \;\Rightarrow\;$ SFLOW $\vdash \mathsf{x} <: \mathsf{poly}$

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-Sovler 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.

```
1  X m(A this) { ...
2    this.f = y; ...
3    X x = new X();
4    return x;
5  }
6  ...
7  x1 = a.m^7();
8  x1.g = 0;
9  safe Y sink = a.f;
10 ...
11 safe X sink1 = a1.m^11();
12 Y x2 = a1.f;
```

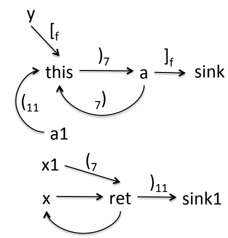


Figure 6. Imprecision in SFlow.

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^i, but it does not flow to sink in context q^j. (More precisely, it would be this.f written in m that would flow to sink.) Also, let ret flow to sink in context q^j, but not in context q^i.

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

$$\mathsf{y} <: q^j \triangleright \mathsf{this} \quad <: \quad q^j \triangleright \mathsf{ret} <: \mathsf{x}$$

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:

$$\mathsf{a} <: q^7 \triangleright \mathsf{this} \quad q^7 \triangleright \mathsf{this} <: \mathsf{a} \quad q^7 \triangleright \mathsf{ret} <: \mathsf{x1} \quad \mathsf{x1} <: q^7 \triangleright \mathsf{ret}$$

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. SFlow creates the following constraints at line 11:

$$\mathsf{a1} <: q^{11} \rhd \mathsf{this} \quad q^{11} \rhd \mathsf{this} <: \mathsf{a1} \quad q^{11} \rhd \mathsf{ret} <: \mathsf{sink1}$$

The last constraint forces ret $<:$ poly.

If this is safe, then due to a1 $<: q^{11} \rhd$ this, a1 is forced to be safe, unnecessarily; if ret is safe, then due to x1 $<: q^7 \rhd$ 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, 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 a1 and x1.

The problem is that in SFlow, 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 $<:$ ret in id), and SFlow is precise. However, when there is no flow from the parameter to the return value, SFlow is imprecise as in Fig. 6.

4.2 Type System

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

The typing rules for SFlowCFL appear in Fig. 7. SFlowCFL augments SFlow with a set of *local constraints* L and a set of rules for the local constraints in L. Constraints in L are of form $\mathsf{x} \dashrightarrow \mathsf{y}$, $\mathsf{y.f} \dashrightarrow \mathsf{x}$ or $\mathsf{x} \dashrightarrow \mathsf{y.f}$, where x and y are local variables in the *same method* m. $L \vdash \mathsf{x} \dashrightarrow \mathsf{y}$ means that there is flow from x to y. All constraints in L must obey the subtyping relation, e.g., if $\mathsf{x} \dashrightarrow \mathsf{y}$ is in L, then $\mathsf{x} <: \mathsf{y}$ must hold in Γ. This is ensured by rule (WELL-FORMED L IN Γ).

Rule (TRANS) adds constraints $\mathsf{x} \dashrightarrow \mathsf{y}$ to L due to transitivity. (TRANS) discovers dependences from formal parameters to return values. We call these dependences *method summary constraints*. Rules (LINEARIZE-TREAD) and (LINEARIZE-TWRITE) add constraints $\mathsf{x} \dashrightarrow \mathsf{y}$ when the adapted field is poly (if f is poly, then $\mathsf{x.f} \dashrightarrow \mathsf{y}$ entails $\mathsf{x} \dashrightarrow \mathsf{y}$ because $\mathsf{x} \rhd \mathsf{f}$ evaluates to x). These rules "erase" field information, akin to the way CFL-Solver "erases" field annotations $[_\mathsf{f}$ and $]_\mathsf{f}$ when f flows into a sink. Just as with CFL-Solver, erasure happens only if f flows into a sink. Intuitively, constraints $\mathsf{x} \dashrightarrow \mathsf{y}$ in L correspond to d-paths in G.

SFlowCFL improves over SFlow because its (TCALL) separates role (1), the propagation of formal-to-return-value dependences (e.g., $\mathsf{p} \dashrightarrow \mathsf{ret}$), which must give rise to actual-to-left-hand-side dependences ($act(\mathsf{p}) \dashrightarrow act(\mathsf{ret})$), 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 $\mathsf{p} \dashrightarrow \mathsf{ret}$. At call $\mathsf{x} = \mathsf{id(y)}$ (TCALL) requires that L entails $\mathsf{y} \dashrightarrow \mathsf{x}$, as expected. The second part creates constraints that link actuals to formals, such as $q_\mathsf{y} <: q_1^i \rhd q_\mathsf{this}$. These constraints capture N-paths and P-paths. Qualifiers q_j^i are distinct, which avoids unnecessary propagation from actuals to left-hand-sides of call assignments.

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

$$\mathsf{a} <: q_1^7 \rhd \mathsf{this} \quad q_1^7 \rhd \mathsf{this} <: \mathsf{a} \quad q_3^7 \rhd \mathsf{ret} <: \mathsf{x1} \quad \mathsf{x1} <: q_3^7 \rhd \mathsf{ret}$$

SFlowCFL enforces the following constraints at line 11:

$$\mathsf{a1} <: q_1^{11} \rhd \mathsf{this} \quad q_1^{11} \rhd \mathsf{this} <: \mathsf{a1} \quad q_3^{11} \rhd \mathsf{ret} <: \mathsf{sink1}$$

(TASSIGN)
$$\frac{L \vdash \mathsf{y} \dashrightarrow \mathsf{x} \qquad ReIm(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow L \vdash \mathsf{x} \dashrightarrow \mathsf{y} \qquad \Gamma \vdash L}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

(TWRITE)
$$\frac{L \vdash \mathsf{x} \dashrightarrow \mathsf{y.f} \qquad ReIm(\mathsf{y.f}) \neq \mathsf{readonly} \Rightarrow L \vdash \mathsf{y.f} \dashrightarrow \mathsf{x} \qquad \Gamma \vdash L}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

(TREAD)
$$\frac{L \vdash \mathsf{y.f} \dashrightarrow \mathsf{x} \qquad ReIm(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow L \vdash \mathsf{x} \dashrightarrow \mathsf{y.f} \qquad \Gamma \vdash L}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

(TCALL)
$$\frac{\begin{array}{c} method(\mathsf{m}) = \mathsf{this}, \mathsf{p} \to \mathsf{ret} \\ L \vdash \mathsf{p} \dashrightarrow \mathsf{ret} \Rightarrow L \vdash act(\mathsf{p}) \dashrightarrow \mathsf{x} \\ L \vdash \mathsf{p} \dashrightarrow \mathsf{p}' \wedge ReIm(\mathsf{p}' \text{ at } i) \neq \mathsf{readonly} \Rightarrow L \vdash act(\mathsf{p}) \dashrightarrow act(\mathsf{p}') \\ typeof(\mathsf{m}) = q_\mathsf{this}, q_\mathsf{p} \to q_\mathsf{ret} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \Gamma(\mathsf{z}) = q_\mathsf{z} \\ q_\mathsf{y} <: q_1^i \rhd q_\mathsf{this} \quad ReIm(\mathsf{this} \text{ at } i) \neq \mathsf{readonly} \Rightarrow q_1^i \rhd q_\mathsf{this} <: q_\mathsf{y} \\ q_\mathsf{z} <: q_2^i \rhd q_\mathsf{p} \quad ReIm(\mathsf{p} \text{ at } i) \neq \mathsf{readonly} \Rightarrow q_2^i \rhd q_\mathsf{p} <: q_\mathsf{z} \\ q_3^i \rhd q_\mathsf{ret} <: q_\mathsf{x} \quad ReIm(\mathsf{x}) \neq \mathsf{readonly} \Rightarrow q_\mathsf{x} <: q_3^i \rhd q_\mathsf{ret} \\ \Gamma \vdash L \end{array}}{\Gamma \vdash i : \mathsf{x} = \mathsf{y.m(z)}}$$

(LINEARIZE-TREAD)
$$\frac{L \vdash \mathsf{x} \dashrightarrow \mathsf{y.f} \quad typeof(\mathsf{f}) = \mathsf{poly}}{L \vdash \mathsf{x} \dashrightarrow \mathsf{y}}$$

(LINEARIZE-TWRITE)
$$\frac{L \vdash \mathsf{y.f} \dashrightarrow \mathsf{x} \quad typeof(\mathsf{f}) = \mathsf{poly}}{L \vdash \mathsf{y} \dashrightarrow \mathsf{x}}$$

(TRANS)
$$\frac{L \vdash \mathsf{x} \dashrightarrow \mathsf{y} \quad L \vdash \mathsf{y} \dashrightarrow \mathsf{z}}{L \vdash \mathsf{x} \dashrightarrow \mathsf{z}}$$

(WELL-FORMED L IN Γ)
$$\frac{L \vdash s \dashrightarrow t \Rightarrow typeof(s) <: typeof(t)}{\Gamma \vdash L}$$

Figure 7. SFlowCFL type system. L is a system of local flow constraints. We extend *typeof* to retrieve the types of variables and expressions; $typeof(\mathsf{x.f}) = \mathsf{x} \rhd \mathsf{f}$ as expected. $act(\mathsf{p})$, used in (TCALL), returns the actual argument corresponding to parameter p.

1: **procedure** SFLOWCFL-INFER
2: **for each** variable x **do** $S(\mathsf{x}) \leftarrow \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$ **end for**
3: **for each** q_j^i at callsite i **do** $S(q_j^i) \leftarrow \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$ **end for**
4: **for each** field f **do** $S(\mathsf{f}) \leftarrow \{\mathsf{tainted}, \mathsf{poly}\}$ **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 $\mathsf{x} <: \mathsf{y} \rhd \mathsf{f}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 1
10: Add $\mathsf{x} <: \mathsf{y}$ into C
11: **else if** c is $\mathsf{x} \rhd \mathsf{f} <: \mathsf{y}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 2
12: Add $\mathsf{x} <: \mathsf{y}$ into C
13: **else if** c is $\mathsf{x} <: \mathsf{y}$ **then** ▷ Case 3
14: **for each** $\mathsf{y} <: \mathsf{z}$ in C **do** Add $\mathsf{x} <: \mathsf{z}$ to C **end for**
15: **for each** $\mathsf{w} <: \mathsf{x}$ in C **do** Add $\mathsf{w} <: \mathsf{y}$ to C **end for**
16: **for each** $\mathsf{w} <: q_j^i \rhd \mathsf{x}$ and $q_k^i \rhd \mathsf{y} <: \mathsf{z}$ in C **do** ▷ Case 4
17: Add $\mathsf{w} <: \mathsf{z}$ to C
18: **end for**
19: **end if**
20: **end for**
21: **until** S remains unchanged
22: **end procedure**

Figure 8. Type inference for SFlowCFL. Lines 2-5 initialize S and C. Cases 1 and 2 add $\mathsf{x} <: \mathsf{y}$ into C because $\mathsf{y} \rhd \mathsf{poly}$ 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 $\mathsf{w} <: q_j^i \rhd \mathsf{x}$ (flow from actual w to formal x) and $q_k^i \rhd \mathsf{y} <: \mathsf{z}$ (flow from return value y to left-hand-side z), and also $\mathsf{x} <: \mathsf{y}$ (flow from formal to return value), Case 4 adds $\mathsf{w} <: \mathsf{z}$. Note that line 4 calls SOLVECONSTRAINT(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until S stays unchanged.

Recall that a and sink1 are safe. this and ret can be poly. $q_1^7 = \mathsf{safe}$, $q_3^7 = \mathsf{tainted}$, $q_1^{11} = \mathsf{tainted}$ and $q_3^{11} = \mathsf{safe}$ 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 $\{\mathsf{tainted}\}$ and $\{\mathsf{safe}\}$, respectively. Fields f are initialized to $S(\mathsf{f}) = \{\mathsf{tainted}, \mathsf{poly}\}$. All other variables and all q_j^i are initialized to the maximal set of qualifiers $\{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

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 $\mathsf{x.f} = \mathsf{y}$ the algorithm adds $\mathsf{y} <: \mathsf{x} \rhd \mathsf{f}$ to C. If y is not readonly, it adds $\mathsf{x} \rhd \mathsf{f} <: \mathsf{y}$ as well. Then the algorithm iterates over constraints $c \in C$ and calls SOLVECONSTRAINT(c). SOLVECONSTRAINT(c) removes *infeasible* qualifiers from the sets of variables in c [12]. Consider constraint c: $\mathsf{y} <: \mathsf{x}$ where $S(\mathsf{y}) = \{\mathsf{tainted}\}$ and $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$. SOLVECONSTRAINT removes poly and safe from $S(\mathsf{x})$, because there does not exist a $\mathsf{y} \in S(\mathsf{y})$ that satisfies $\mathsf{y} <: \mathsf{poly}$ or $\mathsf{y} <: \mathsf{safe}$. In the case that the infeasible qualifier is the last element in $S(\mathsf{x})$, the inference terminates with a *type error*.

The type inference algorithm, SFlowCFL-Infer is presented in Fig. 8. It adds local constraints $\mathsf{x} <: \mathsf{y}$ to C due to transitivity (lines 13-15 mirror rule (TRAINS) in Fig. 7) or due to actual-to-left-hand-side "connections" at method calls (lines 16-18 mirror rule (TCALL) in Fig. 7). It keeps removing infeasible qualifiers for each constraint until (1) it terminates with a type error, indicating potential flow from a source to a sink, or (2) it reaches a fixpoint, meaning that there is no flow from sources to sinks. Sect. 6 gives a detailed example.

When the inference terminates without type errors, we derive a valid typing by picking the *maximal* element of $S(\mathsf{x})$ according to the ranking $\mathsf{tainted} > \mathsf{poly} > \mathsf{safe}$. We call this typing the *maximal typing* T. It is a theorem that T *always type checks* with the rules for SFlowCFL in Fig. 7. The preference ranking gives rise to a ranking over all valid typings. We say that $T' > T''$ if T' has a larger number of tainted variables than T', or T' and T'' have equal number of tainted variables but T' has a larger number of poly variables than T''. It is a theorem that T is the *unique maximal typing*[4] (the proof can be found in [12]). T minimizes the impact of safe through the program, just as CFL-Solver does.

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 $\vdash \mathsf{x} \xrightarrow{d} n \;\Leftrightarrow\; T(\mathsf{x}) <: \mathsf{safe}$
2. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{N} n \;\Leftrightarrow\; T(\mathsf{x}) <: \mathsf{safe}$
3. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{C} n \;\Leftrightarrow\; T(\mathsf{x}) <: \mathsf{poly}$

Theorem 4.2 states that SFlowCFL-Infer terminates with a type error if and only if CFL-Solver discovers a path from a source to a sink. It follows directly from Theorem 4.1.

THEOREM 4.2. CFL-SOLVER $\vdash \mathsf{x} \xrightarrow{C} n$, *where* x *is* tainted *source* $\Leftrightarrow$ *T = error.*

[4] T subsumes 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 n, where $typeof(\mathsf{n}) = q_{\mathsf{this_n}}, q_{\mathsf{p_n}} \rightarrow q_{\mathsf{ret_n}}$ override m, where $typeof(\mathsf{m}) = q_{\mathsf{this_m}}, q_{\mathsf{p_m}} \rightarrow q_{\mathsf{ret_m}}$. SFlow requires

$$\begin{array}{ll} q_{\mathsf{this_m}} <: q_{\mathsf{this_n}} & ReIm(\mathsf{this_n}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{this_n}} <: q_{\mathsf{this_m}} \\ q_{\mathsf{p_m}} <: q_{\mathsf{p_n}} & ReIm(\mathsf{p_n}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{p_n}} <: q_{\mathsf{p_m}} \\ q_{\mathsf{ret_n}} <: q_{\mathsf{ret_m}} & ReIm(\mathsf{ret_m}) \neq \mathsf{readonly} \Rightarrow q_{\mathsf{ret_m}} <: q_{\mathsf{ret_n}} \end{array}$$

The above constraints guarantee that if there is a pair of parameters of overriding method n such that $\mathsf{p_n} <: \mathsf{p'_n}$, then the same subtyping holds for the corresponding pair of parameters of overridden method m: $\mathsf{p_m} <: \mathsf{p'_m}$. (Recall that $\mathsf{p'}$ is usually ret.) For example, suppose that n has statement `return p` but m does not. This entails $\mathsf{p_n} <: \mathsf{ret_n}$. Due to the above constraints, we have $\mathsf{p_m} <: \mathsf{p_n}$ (contravariant arguments) and $\mathsf{ret_n} <: \mathsf{ret_m}$ (covariant return values), which yields $\mathsf{p_m} <: \mathsf{p_{ret}}$. Thus, `return p` in overriding method n will give rise to the expected actual-to-left-hand-side dependence at calls to overriden 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:

$$L \vdash \mathsf{p_n} \dashrightarrow \mathsf{p'_n} \Rightarrow L \vdash \mathsf{p_m} \dashrightarrow \mathsf{p'_m}$$

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 $\textsc{BiEdge}(\mathsf{this_n}, \mathsf{this_m}, \mathsf{this_n}, \mathtt{d}, \mathtt{d})$, $\textsc{BiEdge}(\mathsf{p_n}, \mathsf{p_m}, \mathsf{p_n}, \mathtt{d}, \mathtt{d})$ and $\textsc{BiEdge}(\mathsf{ret_m}, \mathsf{ret_n}, \mathsf{ret_m}, \mathtt{d}, \mathtt{d})$. Thus, CFL-Solver creates edges $\mathsf{this_m} \rightarrow \mathsf{this_n}$, from $\mathsf{p_m} \rightarrow \mathsf{p_n}$ and from $\mathsf{ret_n} \rightarrow \mathsf{ret_m}$ 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 TelephonyManager.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

Line	Code	Constraints	Partial S	New constraints
1	`public class Data {`			
2	`String secret;`		$S(\mathsf{secret}) = \{\mathsf{poly}\}$	
3	`String get(Data this) {return this.secret;}`	$\mathsf{this_{get}} \rhd \mathsf{secret} <: \mathsf{ret_{get}}$	$S(\mathsf{ret_{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	$\mathsf{this_{get}} <: \mathsf{ret_{get}}$
4	`void set(Data this, String p){this.secret = p;}`	$\mathsf{p} <: \mathsf{this_{set}} \rhd \mathsf{secret}$	$S(\mathsf{p}) = \{\mathsf{tainted}, \mathsf{poly}\}$	$\mathsf{p} <: \mathsf{this_{secret}}$
5	`}`			
6	`public class FieldSensitivity2 extends Activity {`			
7	`protected void onCreate(Bundle b) {`			
8	`Data dt = new Data();`		$S(\mathsf{dt}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$	
9	`TelephonyManager tm = (TelephonyManager) getSystemService(''phone'');`			
10	`String sim = tm.getSimSerialNumber();`	$q^{10} \rhd \textbf{tainted} <: \mathsf{sim}$	$S(\mathsf{sim}) = \{\mathsf{tainted}\}$	
11	`dt.set(sim);`	$\mathsf{sim} <: q^{11} \rhd \mathsf{p}$		$\mathsf{sim} <: \mathsf{dt}$
12		$\mathsf{dt} = q^{11} \rhd \mathsf{this_{set}}$	$S(\mathsf{this_{set}}) = \{\mathsf{tainted}, \mathsf{poly}\}$	
13	`SmsManager sms = SmsManager.getDefault();`			
14	`String sg = dt.get();`	$\mathsf{dt} <: q^{14} \rhd \mathsf{this_{get}}$	$S(\mathsf{this_{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	
15		$q^{14} \rhd \mathsf{ret_{get}} <: \mathsf{sg}$		$\mathsf{dt} <: \mathsf{sg}$ TYPE ERROR!
16	`sms.sendTextMessage(''+123'',null,sg,null,null);`	$\mathsf{sg} <: q^{16} \rhd \textbf{safe}$	$S(\mathsf{sg}) = \{\mathsf{safe}\}$	
17	`}`			
18	`}`			

Figure 9. FieldSensitivity2 example refactored from DroidBench. The frame box beside each statement shows the corresponding constraints the statement generates. We omitted uninteresting constraints. The oval boxes show initial propagation. 16 forces sg to be $\{\mathsf{safe}\}$, then 14 forces $\mathsf{ret_{get}}$ to be $\{\mathsf{poly}, \mathsf{safe}\}$ and then 3 forces $\mathsf{this_{get}}$ to be $\{\mathsf{poly}, \mathsf{safe}\}$ and secret to be $\{\mathsf{poly}\}$. 10 forces sim to be $\{\mathsf{tainted}\}$, which in turn forces the parameters p and $\mathsf{this_{set}}$ to be $\{\mathsf{tainted}, \mathsf{poly}\}$. The red frame box in the fourth column (New constraints) shows the added constraints (due to lines 9-19 in Fig. 8). Since field secret is poly, constraint $\mathsf{this_{get}} \rhd \mathsf{secret} <: \mathsf{ret_{get}}$ leads to $\mathsf{this_{get}} <: \mathsf{ret_{get}}$ (Case 2), which in turn leads to $\mathsf{dt} <: \mathsf{sg}$ due to the call at 14 (Case 4). Similarly, $\mathsf{p} <: \mathsf{this_{set}} \rhd \mathsf{secret}$ leads to $\mathsf{p} <: \mathsf{this_{set}}$, which in turn leads to $\mathsf{sim} <: \mathsf{dt}$ due to the call at 11. Since sim is $\{\mathsf{tainted}\}$ and sg is $\{\mathsf{safe}\}$, these constraints cause a TYPE ERROR, detecting the leak. We envision that the type error will be reported at the source or at the sink and will come with this source-sink path: $\mathsf{sim} \xrightarrow{(_{11}} \mathsf{p} \xrightarrow{[_{\mathsf{secret}}} \mathsf{this_{set}} \xrightarrow{)_{11}} \mathsf{dt} \xrightarrow{(_{14}} \mathsf{this_{get}} \xrightarrow{]_{\mathsf{secret}}} \mathsf{ret_{get}} \xrightarrow{)_{14}} \mathsf{sg}$.

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 Fahnrdich [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

[1] Tainting Checker. http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#tainting-checker, 2014.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context-, flow-, field-, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI, to appear*, 2014.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. N. Sheth. TaintDroid : An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[6] M. D. Ernst. Type Annotations specification (JSR 308). http://types.cs.washington.edu/jsr308/, 2012.

[7] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. Technical Report UW-CSE-14-04-02, University of Washington Department of Computer Science and Engineering, Apr. 2014.

[8] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253–263, 2000.

[9] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android application. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[10] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently refactoring java applications to use generic libraries. In *ECOOP*, pages 71–96, 2005.

[11] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST*, pages 273–290, 2012.

[12] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[13] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, pages 879–896, 2012.

[14] W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. In *FASE*, pages 140–154, 2014.

[15] W. Huang, Y. Dong, and A. Milanova. DFlow & DroidInfer: Effective Type-based Taint Analysis for Android. Submitted for publication, 2014. URL http://www.cs.rpi.edu/~milanova/docs/DroidInfer.pdf.

[16] J. Kim, Y. Yoon, and K. Yi. SCANDAL: Static analyzer for detecting privacy leaks in Android applications. In *MoST*, 2012.

[17] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153—-169, 2003.

[18] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, and P. Aldous. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *SPSM*, pages 21–32, 2013.

[19] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, pages 229–240, 2012.

[20] Y. L. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with cfl-reachability. In *CC*, pages 61–81, 2013.

[21] A. Milanova and W. Huang. Composing polymorphic information flow systems with reference immutability. In *FTfJP*, pages 5:1–5:7, 2013.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[24] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, pages 132–145, 1997.

[25] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL reachability. In *POPL*, pages 54–66, 2001.

[26] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

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

[28] A. Sampson, W. Dietl, and E. Fortuna. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[29] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental cfl-reachability summarisation: preliminary experience. In *ASE*, pages 270–273, 2012.

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

[31] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

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

[33] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, pages 167–187, 1996.

[34] G. H. Xu, A. Rountev, and M. Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[35] Z. Yang and M. Yang. LeakMiner: Detect information leakage on Android with static taint analysis. *2012 Third World Congress on Software Engineering*, pages 101–104, Nov. 2012.