FlowCFL: A Framework for Type-based Reachability Analysis in the Presence of Mutable Data

ANA MILANOVA, Rensselaer Polytechnic Institute, milanova@cs.rpi.edu

Reachability analysis is a fundamental program analysis with a wide variety of applications. We present FlowCFL, a framework for type-based reachability analysis in the presence of mutable data. Interestingly, the underlying semantics of FlowCFL is CFL-reachability.

We make three contributions. First, we define a dynamic semantics that captures the notion of flow commonly used in reachability analysis. Second, we establish correctness of CFL-reachability over graphs with inverse edges (inverse edges are necessary for the handling of mutable heap data). Our approach combines CFL-reachability with reference immutability to avoid the addition of certain infeasible inverse edges and we demonstrate empirically that avoiding those edges results in precision improvement. Our formal account of correctness extends to this case as well. Third, we present a type-based reachability analysis and establish equivalence between a certain CFL-reachability analysis and the type-based analysis, thus proving correctness of the type-based analysis.

Additional Key Words and Phrases: CFL-reachability, reference immutability, type-based analysis

1 INTRODUCTION

Reachability analysis detects flow from sources to sinks. It is a fundamental program analysis technique with a wide variety of applications. One prominent application is taint analysis for Android, which detects flow from sensitive sources, such as phone and location data, to untrusted sinks, such as the Internet [Arzt et al., 2014, Ernst et al., 2014, Huang et al., 2015].

In this paper, we study FlowCFL, a framework for type-based reachability analysis. FlowCFL supports two basic type qualifiers, pos (positive) and neg (negative). It permits flow from neg variables to pos ones, but forbids flow from pos variables to neg ones. The principal problem is to decide whether there is flow from a pos variable to a neg one. Our primary contribution is not the FlowCFL system itself; variants of FlowCFL, both graph-reachability-based and type-based have been used in program analysis for a long time. There are publicly available implementations, including our own. Our contribution is the formal treatment of FlowCFL. We formalize the notion of flow in terms of a dynamic semantics and we use the semantics to construct a correctness argument for the static analyses. Another contribution is the interpretation of FlowCFL, a type-based reachability analysis, in terms of the classical theory of Context-Free-Language(CFL)-reachability [Reps, 1998, 2000, Reps et al., 1995].

Standard CFL-reachability analysis has two phases. First, it constructs a graph that represents flow of values from one variable to another; edges are annotated with call and return annotations to model call-transmitted dependences and with field write and field read annotations to model heap-transmitted dependences. Next, the analysis searches for paths with properly matched call/return and write/read annotations. CFL-reachability analysis is a highly precise flow analysis. It has a long history [Reps, 2000, Reps et al., 1995] and it is still actively studied and actively used in program analysis; [Chatterjee et al., 2018, Lu and Xue, 2019, Späth et al., 2019, Xu et al., 2009, Zhang and Su, 2017] are recent works among many other works. An important concept in CFL-reachability analysis is the concept of the inverse edge [Sridharan and Bodík, 2006, Sridharan et al., 2005], which is necessary for the handling of mutable heap data. At assignments, e.g., at $x = y$, the analysis adds the expected forward edge from $y$ to $x$ that represents flow from $y$ to $x$, however, it also adds an inverse edge from $x$ to $y$ thus constructing a bidirectional CFL-reachability graph $G_B$. We discuss inverse edges, the bidirectional CFL-reachability graph, and mutable data in detail in Sect. 4. The concept of the inverse edge, which becomes more involved once we consider
call/return and write/read annotated edges and paths, has not been formalized. The question “Does CFL-reachability over $G_{BI}$ capture all program flows?” has not been answered formally. We consider an answer to this question.

Additionally, we consider a graph that avoids adding certain inverse edges based on knowledge of reference immutability. We denote this graph by $G_{RI}$. Given an immutable reference $x$, there is no need to add inverse paths that originate at $x$. There is substantial precision improvement for reachability over $G_{RI}$ compared to reachability over $G_{BI}$ as demonstrated in earlier work [Milanova and Huang, 2013, Zhang and Su, 2017] and confirmed by experiments we run for this paper. Our formal treatment extends to this case. We answer the following question as well: “Does CFL-reachability over $G_{RI}$ capture all program flows?”.

Returning to FlowCFL, the type-based analysis uses the pos, neg, and poly type qualifiers and a set of typing rules to model reachability. Although not as widespread as CFL-reachability, type-based reachability analysis has been used in a number of existing works, e.g., [Huang et al., 2015, Sampson et al., 2011, Shankar et al., 2001]. Type-based analysis conveniently models reachability problems in different domains, including taint analysis, approximate computing, and secure computation. We establish equivalence between a certain type-based reachability analysis and a certain CFL-reachability analysis over $G_{RI}$, thus establishing correctness of the type-based analysis.

This paper makes the following contributions:

- We present a dynamic semantics that formalizes the notion of flow commonly used in CFL-reachability and type-based reachability.
- We prove that CFL-reachability over $G_{BI}$ captures all run-time flows. Our treatment extends to reachability over $G_{RI}$ which avoids adding certain edges based on knowledge of reference immutability. We present experiments that show substantial precision improvement in taint analysis for Android over $G_{RI}$ compared to analysis over $G_{BI}$. Our experiments are in line with earlier work that has shown the importance of reducing the number of inverse edges [Milanova and Huang, 2013, Zhang and Su, 2017].
- We establish equivalence between a type-based reachability analysis and a CFL-reachability analysis, thus proving correctness of the type-based analysis.

The rest of the paper is organized as follows. Sect. 2 presents an overview of FlowCFL and briefly discusses applications of FlowCFL. Sect. 3 presents the dynamic semantics of flows. Sect. 4 presents CFL-reachability, $G_{BI}$, reference immutability, and the construction of $G_{RI}$. Sect. 5 details the correctness argument. Sect. 6 presents the type-based analysis, and Sect. 7 establishes equivalence between the type-based and CFL-based analyses. Sect. 8 discusses related work and Sect. 9 concludes.

2 OVERVIEW AND APPLICATIONS

2.1 Overview of FlowCFL

In a typical setting, reachability analysis reasons about flow from sources to sinks. FlowCFL assigns type qualifier to variables and fields. There are two basic qualifiers: pos, which denotes sources, and neg, which denotes sinks. We have

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

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 neg variable to a pos one, i.e., a neg variable can flow to a pos one:

\[
\text{neg} \text{ String } n = \ldots;
\]

\[
\text{pos} \text{ String } p = n;
\]
However, it is not allowed to assign a pos variable to a neg one, i.e., a pos variable cannot flow to a neg one:

```java
pos String p = ...;
neg String n = p; // error!
```

Note that this is the natural subtyping. Such subtyping is unsafe in the presence of mutable references [Bank et al., 1997, Sampson et al., 2011] and systems use equality, which is akin to the inverse edges in CFL-reachability. FlowCFL leverages reference immutability (e.g., ReIm [Huang et al., 2012b], Javari [Tschantz and Ernst, 2005]) to allow for safe but limited subtyping.

Once the sources and/or sinks are given, FlowCFL infers qualifiers for the rest of the variables. Roughly, if a source flows to a variable \( x \), then \( x \) is pos; if a variable \( y \) flows to a sink, then \( y \) is neg. If inference fails, i.e., reports error(s), then there may be a leak from a source to a sink. Otherwise, it is guaranteed that there is no flow from a source to a sink.

FlowCFL is context-sensitive (i.e., polymorphic) as illustrated by the following example. We elaborate on context sensitivity in Sects. 4-6.

```java
1 poly String id(poly String p) {  
2     return p;
3 }
4 pos String source = ...;
5 pos String x = id(source);

7 neg String y = ...;
8 neg String sink = id(y);
```

In the above example, the identity function \( id \) is context-sensitive. \( id \) is interpreted as pos in line 5 and it is interpreted as neg in line 8. FlowCFL precisely propagates source to \( x \) but not to sink; it propagates sink back to \( y \) but not to source. A context-insensitive system rejects the program as it merges flow through \( id \) and imprecisely decides that there is flow from source to sink.

From a practical point of view, FlowCFL supports two different settings of the problem. The negative setting assumes a set of initial sink annotations and propagates those sinks backwards, i.e., against the direction of the flow, towards program variables. Unaffected variables remain positive. The more precise the analysis, the fewer variables become neg and a larger number of variables remain pos. The positive setting assumes initial source annotations and propagates those sources forward. The well-known “taint-analysis” problem, which entails annotations on both sources and sinks, can be cast in either of the settings. FlowCFL can run in either the negative or positive setting; if it detects a conflict, i.e., a variable is annotated pos but is inferred neg (or in the positive setting, it is annotated neg but is inferred pos), it reports an error.

### 2.2 Applications

There is a wide variety of applications of FlowCFL. One prominent example, taint analysis for Android reasons about flow of sensitive data (e.g., phone data, location data) to untrusted sinks (e.g., the Internet, Sms texts). [Huang et al., 2015] and [Ernst et al., 2014], among others, describe type-based taint analysis that are instances of FlowCFL. Fig. 1 illustrates taint analysis for Android with FlowCFL. FlowCFL decides that there is flow from positive \( sim \), the device SIM serial number (SSN), to negative \( sg \), the body of the text message. Assuming inference in the negative setting, the analysis determines that \( sim \) is neg, which clashes with the designation of \( sim \) as source and \( sim \)'s pos type. The analysis issues an error that captures that source \( sim \) flows to sink \( sg \).
public class Data {
    String secret;
    void set(String p) {
        this.secret = p;
    }
    String get() {
        return this.secret;
    }
}

public class FieldSensitivity2 extends Activity {
    protected void onCreate(Bundle b) {
        Data dt = new Data();
        TelephonyManager tm = (TelephonyManager)
                getSystemService("phone");
        pos String sim = tm.getSimSerialNumber();
        dt.set(sim);
        SmsManager sms = SmsManager.getDefault();
        neg String sg = dt.get();
        sms.sendTextMessage("+123",null,sg,null,null);
    }
}

Fig. 1. FieldSensitivity2 is rephrased from DroidBench [Arzt et al., 2014, Fritz et al., 2013].
getSimSerialNumber in line 5 in FieldSensitivity2 retrieves sensitive telephony information and its return value is a source. The parameter of sendTextMessage in line 9 is a sink. There is flow from source sim to sink sg through the Data container and FlowCFL reports an error. We note that in actual implementations of taint analysis, there are no annotations in app code only in the Android SDK; we have annotated sim as pos and sg as neg in the above code purely for illustration purposes.

There are many instances of FlowCFL and different instances generally demand different settings. Type systems that underpin approximate computing [Bornholt et al., 2014, Carbin et al., 2013a, Holt et al., 2016, Sampson et al., 2011] are instances of FlowCFL. E.g., EnerJ [Sampson et al., 2011] can be expressed as an instance of FlowCFL in the negative setting. Secure computation, where reachability analysis can partition a program into a secure (and expensive) partition and a plaintext (and inexpensive) partition, is another area of application; the analysis can be expressed as an instance of FlowCFL in the positive setting [Dong et al., 2016]. We have implemented EnerJ [Sampson et al., 2011], Rely [Carbin et al., 2013a], DroidInfer [Huang et al., 2015], and JCrypt [Dong et al., 2016] as instances of FlowCFL (Appendix A describes the instantiations). The wide variety of applications motivates our study of FlowCFL and its connection to CFL-reachability.

2.3 Overview of CFL-reachability
FlowCFL is a type-based analysis but its underlying semantics is the classical CFL-reachability analysis [Reps, 1998, 2000, Reps et al., 1995]. CFL-reachability analysis proceeds in two phases. First, it constructs a flow graph representation of the program. Second, it reasons about reachability over the graph. Throughout the paper, we will work with the example in Fig. 2, which is a rephrase of the FieldSensitivity example in Fig. 1. CFL-reachability constructs the graph shown below. Annotations (i and j) model call-transmitted dependences. For example, edge a \( \xrightarrow{i} p \) represents that at call site 6 in main a flows to parameter p of set, and edge ret \( \xrightarrow{j} b \) represents that at call site 7 ret of get flows to b. Annotations \( w_f \) and \( r_f \) model heap-transmitted dependences. p \( \xrightarrow{w_f} \) this\_set models flow of p into field f of this\_set, and this\_get \( \xrightarrow{r_f} \) ret models read of field f from this\_get into ret.
The principal problem is to decide whether there is flow from one node to another and CFL-reachability analysis makes use of the call/return and write/read annotations to make the decisions. In our example, there is flow from e to b because the call and return annotations, \((\gamma)\) and \((\gamma)\) respectively, match. However, there is no flow from e to d because call annotation \((\gamma)\) and return annotation \((\gamma)\) do not match; they denote two distinct calls. Analogously, field write and field read annotations have to match, there is flow from p to ret because \(w_f\) and \(r_f\) denote a write and a read of the same field \(f\). The analysis decides that there is flow from a to b and from c to d, however there is no flow from a to d or from c to b.

There are two notable points. The first point concerns inverse edges. In the example, there is a forward edge from e to this\(_{set}\) (solid: \(\rightarrow\)) and there is an inverse edge from this\(_{set}\) to e (dashed: \(\rightarrow\)) that reverses the direction of flow and the annotation; call annotation \((\gamma)\) becomes return annotation \((\gamma)\). The forward edge is a natural addition to the graph representing flow of receiver e to this\(_{set}\). The inverse edge, however, is unnatural but it is necessary to discover the path from a to e and then to b as the \textit{mutation}, i.e., update of this\(_{set}\) reverses flow. Standard reachability analysis adds an inverse edge for every forward edge (e.g., [Zhang and Su, 2017], [Lu and Xue, 2019]); in our example, every solid edge would have had a corresponding dashed edge in the standard \(G_{BI}\). Our analysis takes into account reference immutability information and adds only the minimal number of inverse edges necessary to decide flow correctly; the graph shown is the \(G_{RI}\). One key problem we address is to show that \(G_{BI}\), and more interestingly \(G_{RI}\), indeed capture all run-time flows. We define a dynamic semantics that formalizes run-time flows, in our example, the meaning of “p in context of invocation of set in line 6 flows to ret in context of invocation of
get in line 7” (Sect. 3). We proceed to define the construction of $G_{RI}$ and $G_RI$ (Sect. 4) and argue soundness, i.e., that $G_{RI}$ does represent all run-time flows (Sect. 5). The second point concerns paths with interleaved call/return and write/read annotations. For example, the path from $a$ to $b$ involves matching call/return annotations, $(\epsilon_6$ and $\epsilon_a$), as well as $(\epsilon_7$ and $\epsilon_r$, and separately, matching write/read annotations $w_7$ and $r_1$. Exact reasoning over such paths is undecidable [Reps, 2000]. We present a certain approximate reachability analysis over $G_{RI}$ and show that type-based FlowCFL is equivalent to that analysis (Sect. 7). We interpret the seemingly different type-based FlowCFL in terms of CFL-reachability (Sect. 6).

3 DYNAMIC SEMANTICS

In this section we formalize the notion of flow in terms of a dynamic semantics. We restrict our core language to a “named form” in the style of Vaziri et al. [Dolby et al., 2012, Vaziri et al., 2010]. The language models Java with the syntax in Fig. 3, where the results of instantiations, field accesses, and method calls are immediately stored in a variable. Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter.

3.1 Stack Contexts and Chains

Stack contexts describe stack configurations at a point of program execution; as expected, they help formalize run-time local variables, such as the following: “p in context of invocation of set in line 6”. $\langle \text{main}, f_1, f_2...f_n \rangle$ is the stack made up of main, followed by frame identifier $f_1$ corresponding to some callee $m_1$ in main, followed by $f_2$ for some callee $m_2$ in $m_1$, etc, with frame $f_n$ at the top of the stack. Each $f_i$ has a unique identifier—if, say, $m_2$ is called again from the same call site in $m_1$, that would entail a new frame and frame identifier at runtime. We use $A, B, C, ...$ to denote stack contexts. Local variables, naturally, are characterized by their stack context: we write $x^A$ to denote local variable $x$ in context $A$.

In our running example in Fig. 2, we have stack contexts $\langle \text{main}, f_1 \rangle$ and $\langle \text{main}, f_2 \rangle$ where $f_1$ corresponds to the frame of set invoked in line 6, and $f_2$ corresponds to the frame of set invoked in line 8. Variables in set are characterized by their stack context: we write $p^{(\text{main},f_1)}$, this$^{(\text{main},f_1)}$, etc.

The notion of the chain is essential in our treatment. Informally, there is a chain from $x^A$ to $y^B$, denoted by $(x^A, y^B)$, if $x^A$ flows to $y^B$. In Fig. 2 there is a chain from $p^{(\text{main},f_1)}$ to $\text{ret}^{(\text{main},f_1)}$ where $f_2$ is the frame that corresponds to the invocation of get in line 7. Similarly, there are chains $(a^{(\text{main})}, \text{ret}^{(\text{main},f_1)})$, $(a^{(\text{main})}, b^{(\text{main})})$ among others. Chains are represented in $G_{RI}$ by appropriately annotated paths. E.g., chain $(a^{(\text{main})}, \text{ret}^{(\text{main},f_1)})$ is represented by path

\[
\begin{align*}
    a & \mapsto p \mapsto \text{this} \mapsto e \mapsto \text{this} \mapsto \text{ret} \\
    & \mapsto w_7 \mapsto \text{this} \mapsto e \mapsto \text{this} \mapsto \text{ret}
\end{align*}
\]
The unmatched call annotation (γ) represents that a flow into frame f_3 where f_3 maps to call site 7 in the abstract. The string with unmatched call (γ) captures in the abstract the “difference” between contexts (main) and (main, f_3).

The semantics is a standard small-step dynamic semantics extended with the treatment of chains. We write [s'](A, C, S, H) = C', S', H' to model execution of statement s in context A and its effect on chains C, stack S, and heap H. Here C is a map from variables y^B to sets of sources of chains that end at y^B. We say (x^A, y^B) ∈ C iff y^B ∈ Dom(C) and x^A ∈ C(y^B). S and H are the standard maps from variables to objects o (map S), and from object/field tuples o.f to objects o' (map H). Below we discuss the semantics of individual statements.

An assignment x = y in context A records chains that end at x^A. There is a new chain (υ, x^A) ∈ C' for every chain (υ, y^A) ∈ C, and there is a chain (υ, x^A) ∈ C' that accounts for the flow from y to x. The transition on the stack is standard: x^A points to the object that y^A points to and the heap remains the same:

\[\text{ASSIGN} \quad [x = y](A, C, S, H) = C[x^A \leftarrow C(y^A) \cup \{y^A\}], S[x^A \leftarrow S(y^A)], H\]

Field write x.f = y records chains (υ, o.f) and field read y' = x.f references those chains to record (υ, (y')^A):

\[\text{WRITE} \quad [x.f = y](A, C, S, H) = C[o.f \leftarrow C(y^A) \cup \{y^A\}], S, H[o.f \leftarrow o']\]

\[\text{READ} \quad [y' = x.f](A, C, S, H) = C[(y')^A \leftarrow C(o.f)], S[(y')^A \leftarrow o'], H\]

Note that we do not record o.f as a chain source in READ. It is an invariant of C that the values in map C are sets that contain only variables, e.g., x^A. We do record o.f as chain target in WRITE because it serves as an intermediary in the chain from y in x.f = y to y' in y' = x.f. The semantics of chains slides heap objects, just as the static flow graph G_{SI} does (recall the graph in Sect. 2.3). The goal is to establish a connection between the concrete domain of chains and the abstract domain of annotated paths in the flow graph.

Allocation x = new o creates the trivial chain (x^A, o^A):

\[\text{ALLOC} \quad [x = \text{new } o](A, C, S, H) = C[x^A \leftarrow \{x^A\}], S[x^A \leftarrow o], H[o.f \leftarrow \text{null}]\]

where o is a fresh object

Calls and returns are standard. A call entails a fresh frame identifier f, which is appended to context A to form the new context A@f; calls record chains that reflect the standard flow from actuals to formals:

\[\text{CALL} \quad [x = \text{y.m(z)}](A, C, S, H) = C[\text{this}^{A@f} \leftarrow C(y^A) \cup \{y^A\}], S[\text{this}^{A@f} \leftarrow S(y^A)])[p^{A@f} \leftarrow C(z^A) \cup \{z^A\}], H\]

where f is a fresh frame identifier

A return from context A@f creates chains with target x^A; these chains are due to the standard flow from ret^{A@f} to the left-hand side of the return assignment x^A:

\[\text{RET} \quad [x = \text{y.m(z)}](A@f, C, S, H) = C[x^A \leftarrow C(\text{ret}^{A@f}) \cup \{\text{ret}^{A@f}\}], S[x^A \leftarrow S(\text{ret}^{A@f})], H\]

The following lemma allows us to express the points-to relation entailed by S and H in terms of the reachability relation entailed by C. If a variable x^A points to some object o, then there is a chain (w^B, x^A) in C where w^B is the local variable at the left-hand side of the allocation site of o. If we have x.f = y in context A followed by y' = x'.f in context A', where x and x' point to the same object o, then there is flow from y to y'. The flow is expressed via paths that capture chains...
which abstracts the chain from w to x and an inverse path from x to w. The inverse path x \rightsquigarrow w "combines" with the path w \rightsquigarrow x' which leads to a path x \rightsquigarrow x' and subsequently y \rightsquigarrow y'. Recall Fig. 2 and the accompanying graph:

\[
\text{this}_\text{set} \arr y \rightarrow e \text{ and } e \arr y \rightarrow \text{this}_\text{get} \text{ give rise to } p \arr w \rightarrow \text{this}_\text{set} \arr y \rightarrow e \arr y \rightarrow \text{this}_\text{get} \arr y \rightarrow \text{ret}
\]

which abstracts the chain from \(p^{(\text{main}, f_3)}\) to \(\text{ret}^{(\text{main}, f_3)}\). We elaborate later in the paper.

**Lemma 3.1.** For every state \(C, S, H\) and every object \(o \in H\)

- \(S(x^A) = o \implies (w^B, x^A) \in C\) and
- \(H(o'.f) = o \implies (w^B, o'.f) \in C\)

where \(w = \text{new } C\) in context \(B\) is the creation site of \(o\).

**Proof.** Given transition \([s](A, C, S, H) = C', S', H'\) we show, via case-by-case analysis, that if the lemma holds on \(C, S, H\) then it holds on \(C', S', H'\). Consider ALLOC. The only change is in \(S\) where \(S'(x^A)\) now points to a fresh heap object \(o\). \(C'\) adds \((w^A, w^A)\) which establishes the lemma. The remaining points-to relations have not changed and the inductive hypothesis entails the lemma. The rest of the statements follow analogously.

\[\square\]

### 3.2 Operations on Contexts

Next we define several useful operations on stack contexts. \(A - B\) is defined when \(B\) is a prefix of \(A\); it removes \(B\) from \(A\). \(A \leq B\) is true if and only if \(A\) is a prefix of \(B\). \(\Delta AB\) denotes the "difference" between context \(A\) and context \(B\). At the level of the dynamic semantics, \(\Delta AB\) is defined as the tuple \((A - D, B - D)\), where \(D\) is the longest common prefix of \(A\) and \(B\). Such a prefix clearly exists, in the worst case it is main.

Returning to Fig. 2, consider the flow from \(p\) in context \(\langle \text{main}, f_2 \rangle\) of set, to \(\text{ret}\) in context \(\langle \text{main}, f_3 \rangle\) of get (recall that \(f_3\) is the frame invoked in line 7). We have:

\(\Delta\langle \text{main}, f_2 \rangle \langle \text{main}, f_3 \rangle = (\langle f_2 \rangle, \langle f_3 \rangle)\)

Informally, the first term in the tuple is the sequence of \textit{returns} and the second term is the sequence of \textit{calls} that happen when state transitions from \(A\) to \(B\). In the example, stack state transitions from \(\langle \text{main}, f_2 \rangle\) to \(\langle \text{main}, f_3 \rangle\), by first returning from \(f_2\) into \(\text{main}\) then calling into \(f_3\) from main.

In Sect. 5 we define an abstraction function over stack contexts and \(\Delta AB\) that helps establish that for every chain from \(x^A\) to \(y^B\), \(G_{BI}\) and \(G_{RI}\) contain appropriately annotated paths from \(x\) to \(y\). As stated earlier, a key difficulty arises in the reasoning about inverse edges.

### 4 CFL-REACHABILITY

Sect. 4.1 describes the construction of \(G_{BI}\). Sect. 4.2 argues that there is inherent imprecision in \(G_{BI}\). Sect. 4.3 discusses reference immutability and Sect. 4.4 describes the construction of \(G_{RI}\) based on knowledge of reference immutability.

#### 4.1 Bidirectional Flow Graph \(G_{BI}\)

As it is customary for CFL-reachability, we build a static \textit{flow graph} that represents data dependences between variables. The nodes in the graph are (context-insensitive) program variables, e.g., \(x, y, \text{this}\). The edges capture flow from one variable to another and paths capture dynamic chains as defined in Sect. 3. The standard approach in the presence of mutable data is to build a \textit{bidirectional} flow graph (as in [Chatterjee et al., 2018, Sampson et al., 2011, Shankar et al., 2001, Späth et al.,]...
2019, Sridharan and Bodik, 2006, Xu et al., 2009, Zhang and Su, 2017] among other works) where inverse edges handle updates safely. We call this graph $G_{BI}$. Below we describe the semantics of $G_{BI}$ construction. Solid arrows $\rightarrow$ denote forward edges, and dashed arrows $\rightarrow$ denote inverse edges.

An assignment statement contributes direct (i.e., intraprocedural) edges as follows:

\[
\text{ASSIGN}\quad [x = y](G_{BI}) = G_{BI} \cup \{ y \overset{d}{\rightarrow} x \} \cup \{ x \overset{d}{\rightarrow} y \}
\]

A field write statement $x.f = y$ contributes a forward edge from $y$ to $x$ annotated with $w_f$ and an inverse edge from $x$ to $y$ annotated with $r_f$:

\[
\text{WRITE}\quad [x.f = y](G_{BI}) = G_{BI} \cup \{ y \overset{w_f}{\rightarrow} x \} \cup \{ x \overset{r_f}{\rightarrow} y \}
\]

The meaning of the forward edge is that $y$ flows (is written) into field $f$ of $x$. The corresponding inverse edge reverses the direction of the flow and the field annotation, denoting that field $f$ of $x$ is read into $y$. Similarly, a field read statement $y' = x'.f$ contributes

\[
\text{READ}\quad [y' = x'.f](G_{BI}) = G_{BI} \cup \{ x' \overset{r_f}{\rightarrow} y' \} \cup \{ y' \overset{w_f}{\rightarrow} x' \}
\]

The following example illustrates once again the need for inverse edges. From now on, we will underline sinks in the graphs to improve readability.

\[
\begin{array}{l}
1. \quad x = y; \\
2. \quad A a = \ldots ; \\
3. \quad x.f = a; \\
4. \quad \text{neg} A \ b = y.f;
\end{array}
\]

In reverse edge $x \overset{d}{\rightarrow} y$ is necessary to establish the path from a to b.

A method call (method entry) creates the expected forward call edges from actual arguments to formal parameters and the inverse return edges:

\[
\text{CALL}\quad [i : x = y.m(z)](G_{BI}) = G_{BI} \cup \{ y \overset{t}{\rightarrow} \text{this} \} \cup \{ z \overset{t}{\rightarrow} p \} \cup \{ \text{this} \overset{t}{\rightarrow} y \} \cup \{ p \overset{t}{\rightarrow} z \}
\]

The standard CFL-reachability annotation ($t$ marks call entry at call site $i$. A method return (exit) creates a return edge from the return value to the left-hand-side of the call assignment, plus the inverse call edge:

\[
\text{RET}\quad [i : x = y.m(z)](G_{BI}) = G_{BI} \cup \{ \text{ret} \overset{t}{\rightarrow} x \} \cup \{ x \overset{t}{\rightarrow} \text{ret} \}
\]

The CFL-reachability problem is to decide whether there is a path from $x$ to $y$ in $G_{BI}$ with properly matched call/ret annotations, and properly matched write/read annotations. Note the arbitrary interleaving of ($t$ and $w, r$ annotations. Due to recursion in both call-transmitted and heap-transmitted dependences, the CFL-reachability problem is undecidable [Reps, 2000] and analyses have to adopt approximations. One approximation essentially replaces all ($t$ and $r$) with $d$ annotations and leaves all $w, r$ annotations, thus treating call-transmitted dependences context-inensively and heap-transmitted dependences fully precisely. This approach is known as CIFS (context-insensitive, field-sensitive) CFL-reachability [Zhang and Su, 2017]. Another approximation replaces all $w_f$ and $r_f$ annotations with $d$ thus treating call-transmitted dependences fully precisely and heap-transmitted dependences approximately. This approach is known as CSFI (context-sensitive, field-insensitive) CFL-reachability. Consider the CR and PG context-free grammars in Fig. 4. CSFI amounts to PG-reachability, i.e., if the path from $x$ to $y$ is in the language $L(PG)$, then $y$ is PG-reachable from $x$. Analogously, CIFS amounts to CR-reachability, i.e., if the path from $x$ to $y$ is in $L(CR)$, then $y$ is CR-reachable from $x$. There are many approximations in the literature.
Fig. 4. The CR grammar in (a) captures well-formed call-transmitted paths. $R$ captures strings with outstanding (R)eturn edges, e.g., $(1 \ d \ 1)$. $C$ captures strings with outstanding (C)all edges, e.g., $(1 \ M \ 1)$. $M$ captures same-level paths, i.e., paths with matching call and return edges, e.g., $(1 \ d \ 1)$. The PG grammar in (b) captures heap-transmitted paths. $P$ captures strings with outstanding [G]etfield (i.e., field read) edges, e.g., $w_f \ d \ r_f$. $P$ captures strings with outstanding [P]utfield (i.e., field write) edges, e.g., $w_f \ d$. Finally, $B$ captures strings with matching field write and field read edges, e.g., $w_f \ d \ r_f$.

4.2 Imprecision in $G_{BI}$

Bidirectionality of $G_{BI}$ causes imprecision. Consider the example:

```
1 if (c) {
2   x = a;
3 }
4 else {
5   x = b;
6 }
```

The two forward edges have corresponding inverse edges, creating paths from $x$ to $b$ and from $a$ to $b$. If $b$ is negative, then both $x$ and $a$ spurious become negative. The imprecision propagates, “polluting” variables throughout the program.

[Zhang and Su, 2017] report that linear conjunctive reachability over $G_{BI}$, a novel highly-precise CFL-reachability technique, achieves only modest improvement compared to CSFI over $G_{BI}$; they conjecture that this is due to the bidirectionality of $G_{BI}$. In contrast, the technique achieves substantial precision improvement over $G_{RI}$ [Zhang and Su, 2017]. (Recall that $G_{RI}$ is the strict subgraph of $G_{BI}$ that avoids certain inverse edges; we describe reference immutability and the construction of $G_{RI}$ in Sect. 4.3 and Sect. 4.4.) Similarly, [Milanova and Huang, 2013] report substantial negative impact of bidirectionality on taint analysis—a taint analysis that removes infeasible edges based on reference immutability information infers 20% to 79% fewer negative variables compared to a taint analysis on the bidirectional graph. (Recall that the goal of taint analysis is to propagate negative sinks to as few program variables as possible.)

We conducted experiments using the implementation and benchmarks of DroidInfer [Huang et al., 2015] that are made publicly available with the artifact of DroidInfer.¹ We ran the analysis on 77 Android apps from the artifact, excluding apps that crashed and apps that contained 0 sources or 0 sinks. We ran the taint analysis over $G_{BI}$ and over $G_{RI}$. Analysis over $G_{RI}$ reduced the number of reported errors by 41% on average per app compared to $G_{BI}$. An error essentially corresponds to a (source, sink) pair, or in other words, to a report of flow from source to sink; the analysis proved

¹The artifact is publicly available at https://www.cs.rpi.edu/~dongy6/issta-artifact-2015/issta-artifact-2015.zip; the code is available at https://github.com/proganalysis/type-inference.
A key goal of this work is to formalize the notion of the inverse edge and understand the role of inverse edges in removing infeasible inverse edges benefits analysis precision.

Inverse edges capture bidirectionality of aliasing. Suppose reference \( x \) flows to \( x' \). Then for all fields \( f \), \( x.f \) and \( x'.f \) are aliases. If there is a \textit{write} into \( x'.f \) then we should be able to \textit{read} that value out of \( x.f \) and the inverse path from \( x' \) to \( x \) enables that. However, if \( x' \) is an immutable reference, then there is no need for the inverse path from \( x' \) to \( x \).

### 4.3 Reference Immutability

A key goal of this work is to formalize the notion of the inverse edge and understand the role of reference immutability in removing infeasible inverse edges. Reference immutability [Huang et al., 2012b, Milanova, 2018, Tschantz and Ernst, 2005] ensures that a \textit{readonly} (also called immutable) reference cannot be used to mutate the state of the object it refers to, including its transitive state. For example, \( x \) is not readonly in \( y = x; y.f = z; \) because it is used in a way that leads to a mutation of the object it references; similarly \( x \) is not readonly in \( y.f = x; z = y; w = z.f; w.g = 10; \).

Reference immutability semantics is typically described in terms of a type system, most notably Java [Tschantz and Ernst, 2005] and ReIm [Huang et al., 2012b]. Recent work has shown that it can be described in terms of CFL-reachability as well [Milanova, 2018]. In this paper, we largely follow the CFL-reachability interpretation of ReIm given in [Milanova, 2018].

In the style of Sect. 4.1, we analyze each program statement and build a \textit{reference immutability graph} \( G \) then decide immutability/mutability of references based on reachability over \( G \). Informally, \( x \) is mutable if it reaches a variable that is updated, such as \( y \) and \( w \) above. An assignment statement contributes the following forward edge to \( G \). Notably, there are no inverse edges in reference immutability:

\[
\text{ASSIGN } \ [x = y]_G = G \cup \{ y \rightarrow x \}
\]

Similarly to Sect. 4.1 a method call creates call and return forward edges as expected:

\[
\text{CALL/RET } \ [i : x = y.m(z)]_G = G \cup \{ y \xrightarrow{ \text{this} } \} \cup \{ z \xrightarrow{ p } \} \cup \{ \text{ret} \xrightarrow{ } x \}
\]

A difference with Sect. 4.1 arises in the handling of heap-transmitted dependences. Together, a pair of field write \( x.f = y \) and field read \( y' = x'.f \) contribute the following edges

\[
\text{WRITE/READ } \ [x.f = y, \ y' = x'.f]_G = G \cup \{ y \xrightarrow{ } x.f \xrightarrow{ a } x'.f \xrightarrow{ d } y' \} \cup \{ x' \xrightarrow{ d } x'.f \}
\]

The first set of edges creates a path from \( y \) to \( y' \). Here \( x.f \xrightarrow{ a } x'.f \) is an \textit{approximate edge}. In terms of Reps’ terminology [Reps, 2000], the reference immutability semantics models heap-transmitted dependences approximately. The approximation comes from the fact that regardless of whether \( x.f \) and \( x'.f \) are aliases, the semantics propagates \( y' \) back to \( y \). If there is an update of \( y' \), e.g., \( y'.g = 5 \), \( y \) will be determined mutable. This is precisely what makes inverse edges unnecessary here. A field read contributes an additional edge \( x' \xrightarrow{ d } x'.f \) needed to capture mutation to transitive state. If there is a mutation of \( y' \), this last edge propagates the mutation back to \( x' \).

An \textit{update} is a node \( y \) such that there is a write statement of the form \( y.f = z \). Essentially, we are interested if there is an “appropriately annotated” path in \( G \) from a reference \( x \) to an update node. For example, consider the code and its corresponding graph \( G \). \( i.d \) is the standard identity function:

\[
i : x = \text{id}(y); z = x; w = z.f; w.g = 10; \quad \Rightarrow \quad y \xrightarrow{ \text{this} } p \xrightarrow{ ret } x \xrightarrow{ d } z \xrightarrow{ d } z.f \xrightarrow{ d } w
\]
There is a path from $y$ to the update $w$ with properly matched parenthesis which means that $y$ is a mutable reference; the object $o$ that $y$ refers to is modified through $y$ as the assignment of $y$ to parameter $p$ of $id$ leads to the mutation $o.f.g = 10$

A call-transmitted path contains no approximate edges and it is well-formed in CR, i.e., its annotations form a string in L(CR) (recall Fig. 4(a)). A heap-transmitted path is made up of call-transmitted paths interleaved with approximate edges. Let $U$ denote the set of all call-transmitted and heap-transmitted paths to updates in $G$. We break paths in $U$ into 2 categories: (1) $M/C$-paths and (2) $R$-paths. A path in $U$ is an $M/C$-path if and only if the annotations on the leading, i.e., first, call-transmitted path form a string in the language described by $M$ (i.e., calls and returns balance out), or they form a string in the language described by $C$ (i.e., outstanding calls). For example, $e \xrightarrow{\ell} \text{this}_{\text{set}}$ is a $C$-path. A path in $U$ is an $R$-path if and only if the edge annotations on the leading call-transmitted path form a string in $R$, e.g., $\text{ret} \xrightarrow{\ell} b$ is an $R$-path. We consider that there is a trivial path from each node to itself, so an update node is mutable.

(1) Examples of $M/C$-paths, following the graph in Sect. 2.3, include (assuming $b$ is an update):

$$
e \xrightarrow{\ell} \text{this}_{\text{get}} \xrightarrow{d} \text{this}_{\text{get}}.f \xrightarrow{d} \text{ret} \xrightarrow{\ell} b$$ (leading call-transmitted path is an $M$-path)

and

$$a \xrightarrow{\ell} p \xrightarrow{d} \text{this}_{\text{set}}.f \xrightarrow{a} \text{this}_{\text{get}}.f \xrightarrow{d} \text{ret} \xrightarrow{\ell} b$$ (leading call-transmitted path is a $C$-path)

(2) examples of $R$-paths include (again assuming $b$ is an update):

$$\text{this}_{\text{get}} \xrightarrow{d} \text{this}_{\text{get}}.f \xrightarrow{d} \text{ret} \xrightarrow{\ell} b$$

Reference immutability [Huang et al., 2012b, Milanova, 2018, Tschantz and Ernst, 2005] classifies variables as follows:

- $x$ is mutable if there is an $M/C$-path from $x$ to an update
- $x$ is poly if there is no $M/C$-path but there is an $R$-paths from $x$ to an update
- $x$ is readonly if there is neither $M/C$-path nor $R$-path from $x$ to an update

In the above examples $e$ and $a$ are both mutable and $\text{this}_{\text{get}}$ and $\text{ret}$ are poly, due to the $R$-paths to the update. If we removed the assumption that $b$ is an update, then $e$, $a$, $\text{this}_{\text{get}}$ and $\text{ret}$ above would be readonly. Intuitively, an $M/C$-path from $x$ unequivocally makes $x$ mutable—mutation is immediate or within the immediate call. An $R$ path does not necessarily make $x$ mutable; it is mutable in the context of the $R$-path, but it may be readonly in the context of other paths. For example, mutable $b = e.get()$ makes $\text{this}_{\text{get}}$ mutable in the context of this path (precisely the $R$-path above), however, readonly $d = g.get()$ leaves $\text{this}_{\text{get}}$ readonly in the context of this different path: $\text{this}_{\text{get}} \xrightarrow{d} \text{this}_{\text{get}}.f \xrightarrow{d} \text{ret} \xrightarrow{\ell} d$.

Lastly, we summarize the adaptation operation [Dietl et al., 2011, Huang et al., 2012b] which plays a role in the theorem that proves soundness of CFL-reachability over $G_{RI}$. Viewpoint adaptation, written as $x \triangleright p$, interprets the immutability type of $p$ in the context of $x$. We simplify the adaptation notation to work on variables rather than immutability qualifiers. $x \triangleright p$ refers to the immutability types of $x$ and $p$, not to the variables themselves.

$$x \triangleright$ readonly = readonly $x \triangleright$ mutable = mutable $x \triangleright$ poly = $x$

A readonly or mutable $p$ remains as is, regardless of the context $x$. A poly $p$ takes the type of $x$, which makes adaptation interesting. In reference immutability, left-hand sides $x$ of call assignments
serve as contexts of adaptation. This is because, intuitively, there are multiple paths from a poly variable \( p \) in \( m \), one through each one of the left-hand sides of calls to \( m \); the mutability status of each path is determined by the left-hand side of the call. If \( i: x = m(z) \) is such that \( x \) is mutable, then \( p \) is mutable at \( i \). Returning to the examples above, at mutable \( b = e().get() \) we have \( b \triangleright this_{get} = \text{mutable} \) and \( b \triangleright ret = \text{mutable} \) reflecting that \( this_{get} \) and \( ret \) are mutable at the call to \( get \) at 7 because the left-hand side, i.e., the context \( b \) is mutable. In contrast, at readonly \( d = g().get() \) we have \( d \triangleright this_{get} = \text{readonly} \) and \( d \triangleright ret = \text{readonly} \).  

We generalize adaptation to a sequence of contexts, e.g., \( x_i \triangleright x_j \triangleright x_k \triangleright ret \) adapts \( ret \) in a larger context. Suppose \( ret \) is poly, and left-hand-sides \( x_k \) at call \( k \) and \( x_j \) at call \( j \) are poly, however, \( x_i \) at the outermost call site \( i \) is readonly; \( ret \) in context \( x_i \triangleright x_j \triangleright x_k \) is readonly.

1. \( A \text{id0}(A p0) \{ \text{ret0} = p0; \} \)
2. \( A \text{id1}(A p1) \{ \text{ret1} = \text{id0}(p1); \} \)
3. \( A \text{id2}(A p2) \{ \text{ret2} = \text{id1}(p2); \} \)
4. \( \ldots \)
5. \( \text{mutable} b = \text{id2}(a); \)
6. \( b.f = z; \)
7. \( \text{readonly} d = \text{id2}(c); \)

In the above example, \( \text{ret0}, \text{p0}, \text{ret1}, \text{p1}, \text{ret2}, \) and \( \text{p2} \) are all poly due to the \( R \)-paths to update \( b \). In context \( 7 \) \( \text{ret0} \) is interpreted as \( b \triangleright \text{ret2} \triangleright \text{ret1} \triangleright \text{ret0} = \text{mutable} \triangleright \text{poly} \triangleright \text{poly} \triangleright \text{poly} = \text{mutable} \). \( b \triangleright \text{ret2} \triangleright \text{ret1} \) abstracts the stack context of \( \text{id0} \) that line 7 initiates.

### 4.4 Flow Graph \( G_{RI} \)

We use reference immutability to build a new flow graph \( G_{RI} \) without certain infeasible inverse edges. In summary, explicit and implicit assignments forgo inverse edges if the left-hand-side of the assignment is readonly as illustrated by the rule for assignment statement.

\[
\text{ASSIGN} \quad [x = y](G_{RI}) = \begin{cases} 
G_{RI} \cup \{ y \overset{d}{\rightarrow} x \} \cup \{ x \overset{d}{\rightarrow} y \} & \text{if } x \text{ is not readonly} \\
G_{RI} \cup \{ y \overset{d}{\rightarrow} x \} & \text{otherwise}
\end{cases}
\]

The rules for WRITE, READ, CALL and RET are analogous. CALL \( i: x = y.m(z) \) adds inverse edge this \( \overset{i}{\rightarrow} y \) when \( x \triangleright \) this \( \neq \) readonly, and it adds \( p \overset{\triangleright}{\rightarrow} z \) when \( x \triangleright p \neq \) readonly. RET adds \( x \overset{\triangleright}{\rightarrow} \) ret when \( x \triangleright \text{ret} \neq \) readonly.

Consider the example in Fig. 5. \( p \) and \( y \) are updates and therefore mutable, and \( z \) is mutable as well due to the \( C \)-path to update \( p: z \overset{\overset{\triangleleft}{\rightarrow}}{\rightarrow} p \). \( x \) and \( y \) are poly due to the \( R \)-path \( x \overset{d}{\rightarrow} \text{ret} \overset{\triangleleft}{\rightarrow} y \). However, \( w, a, b, \) as well as fields \( f \) and \( g \) are readonly, as there is neither \( M/C \)-path nor \( R \)-path to an update. The rules above entail only 3 inverse edges: (1) \( p \overset{\overset{\triangleleft}{\rightarrow}}{\rightarrow} z \), (2) \( y \overset{\overset{\triangleleft}{\rightarrow}}{\rightarrow} \text{ret} \), and (3) \( \text{ret} \overset{\overset{\triangleleft}{\rightarrow}}{\rightarrow} x \), precisely the edges needed to capture the path from \( a \) to \( b \).

Our key result is that paths in \( G_{RI} \) with properly matched \( w/r \) and call/ret annotations capture all chains as defined in Sect. 3.

---

2Standard viewpoint adaptation [Dietl et al., 2011] uses the receiver as context of adaptation, e.g., in \( x = y.m(z) \), the context of adaptation is \( y \). The use of left-hand side \( x \) is non-standard and reflects the specific semantics of reference immutability; [Huang et al., 2012b] elaborates on this.
We define the concatenation operation over abstract differences as follows:

\[
\alpha \Delta \beta = \langle i_1, i_2, \ldots, i_m \rangle
\]

To prove soundness of reachability over \(G_{RI}\), we first define the abstraction function \(\alpha(A)\) over stack contexts \(A\):

\[
\alpha((\text{main}, f_1, f_2, \ldots, f_n)) = \langle i_1, i_2, \ldots, i_n \rangle
\]

where \(i_1, i_2, \ldots, i_n\) are the static call sites that triggered frames \(f_1, f_2, \ldots, f_n\). \(\alpha(A)\) extends to partial contexts \(A\) as follows:

\[
\alpha((f_2, \ldots, f_n)) = \langle i_2, \ldots, i_n \rangle
\]

We define the abstract difference, \(\alpha(\Delta AB)\), as the tuple \((\alpha(A - D), \alpha(B - D))\) where \(D\) is the longest common prefix of \(A\) and \(B\). We will denote these tuples as \((\text{ret}, \text{call})\) as \(\alpha(A - D)\) abstracts a certain \textit{return sequence} and \(\alpha(B - D)\) abstracts a certain \textit{call sequence} as we discussed in Sect. 3.2.

We define the concatenation operation over abstract differences as follows:

\[
\text{ret}_1, \text{call}_1 \oplus \text{ret}_2, \text{call}_2 = \begin{cases} 
\text{ret}_1, (\text{call}_1 - \text{ret}_2) + \text{call}_2 & \text{if } \text{ret}_2 \text{ is a suffix of } \text{call}_1 \\
\text{ret}_1 + (\text{ret}_2 - \text{call}_1), \text{call}_2 & \text{if } \text{call}_1 \text{ is a suffix of } \text{ret}_2 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

In the above, we overload \textit{minus} – to subtract a suffix, not just a prefix of a string and + is just standard string concatenation. Consider \(\alpha(\Delta AB) = (\alpha(A - D), \alpha(B - D)) = (\text{ret}_1, \text{call}_1)\) and \(\alpha(\Delta BC) = (\alpha(B - E), \alpha(C - E)) = (\text{ret}_2, \text{call}_2)\). If we think of \textit{ret} strings as strings of closing parentheses, i.e., \(\text{ret} = \langle i_1, i_2, \ldots, i_n \rangle = i_n \ldots i_2 \rangle i_1\), and of \textit{call} strings as strings of opening parentheses, i.e., \(\text{call} = \langle j_1, j_2, \ldots, j_m \rangle = j_m \ldots j_2 \rangle j_1\), then concatenation cancels out call annotations \(\langle k \rangle\) and return annotations \(\langle \rangle\). For the remainder of this section we will interpret \(\alpha(\Delta AB)\) to mean both the tuple \((\text{ret}, \text{call})\) as defined above and the string \(i_n \ldots i_2 \rangle i_1\). In the abstract domain, graph \(G_{RI}\), we will be looking at paths from \(x\) to \(y\); a chain \((x^A, y^B)\) will map into a path from \(x\) to \(y\) in \(G_{RI}\) such that the string of unmatched call and return annotations on that path is exactly \(\alpha(\Delta AB)\). If \(\text{call}_1 = \alpha(B - D)\) is longer than \(\text{ret}_2 = \alpha(B - E)\), then \(\alpha(B - D)\) cancels out \(\alpha(B - E)\) and the outstanding call annotations are prepended onto \(\alpha(C - E)\). Analogously, if \(\alpha(B - E)\) is longer than \(\alpha(B - D)\) then \(\alpha(B - E)\) cancels out \(\alpha(B - D)\), and the outstanding return annotations are appended.

---

**Fig. 5.** An example contrasting \(G_{RI}\) and \(G_{BI}\). \(G_{RI}\) retains the path from \(a\) to \(b\) but avoids multiple infeasible paths from \(G_{BI}\), e.g., the path from \(b\) to \(a\).
to \( ret_1 = \alpha(A - D) \). If neither \( call_1 \) cancels out \( ret_2 \), nor \( ret_2 \) cancels \( call_1 \), then concatenation is undefined because the strings represent distinct runtime contexts.

As an example, return to Fig. 2. \( p^{(\text{main}, f_1)} \) flows to \( ret^{(\text{main}, f_3)} \), and we are interested in the abstract difference \( \alpha(\Delta(\text{main}, f_1)\langle \text{main}, f_3 \rangle) \) which is \( ((6), (7)) \) or just the string \( \bar{\eta}_6(7) \). Similarly, since \( ret^{(\text{main}, f_2)} \) flows to \( b^{(\text{main})} \) we are interested in \( \alpha(\Delta(\text{main}, f_3)\langle \text{main} \rangle) \) which is just \( \eta_7 \), with an empty call sequence. \( \bar{\eta}_6(7 + \gamma \eta_7) \) equals \( \bar{\eta}_7 \).

The concatenation lemma below states that if we have the strings \( \alpha(\DeltaAB) \) and \( \alpha(\DeltaBC) \) their concatenation as defined above produces \( \alpha(\DeltaAC) \), precisely the abstraction of \( \DeltaAC \).

**Lemma 5.1.** \( \alpha(\DeltaAB) \oplus \alpha(\DeltaBC) = \alpha(\DeltaAC) \)

**Proof.** Let \( \alpha(\DeltaAB) = (\alpha(A-D), \alpha(B-D)) = (ret_1, call_1) \) and let \( \alpha(\DeltaBC) = (\alpha(B-E), \alpha(C-E)) = (ret_2, call_2) \). Here \( D \) is the longest common prefix of \( A \) and \( B \) and \( E \) is the longest common prefix of \( B \) and \( C \). There are two cases, (1) \( D \leq E \) and (2) \( E \leq D \). We argue case (1), case (2) is analogous. Since \( D \leq E \), it follows that \( ret_2 = \alpha(B - E) \) is a suffix of \( call_1 = \alpha(B - D) \). By the definition of concatenation above we have

\[
\alpha(\DeltaAB) \oplus \alpha(\DeltaBC) = (\alpha(A - D), \alpha(C - E) + \alpha(E - D)) = (\alpha(A - D), \alpha(C - D))
\]

It remains to make the argument that \( D \) is the longest common prefix of \( A \) and \( C \) which is true because \( E - D \) does not overlap with \( A - D \) or otherwise the longest common prefix of \( A \) and \( B \) would have been longer than \( D \). Therefore \( (\alpha(A-D), \alpha(C-D)) = \alpha(\DeltaAC) \) which establishes the statement.

Our main theorem, Theorem 5.2, shows that every chain \((x^A, y^B)\) in \( C \) is represented by an appropriately annotated path from \( x \) to \( y \) in \( G_{RI} \). We write \( x \sim_{\alpha(\DeltaAB)} y \) to denote the existence of a path from \( x \) to \( y \) in \( G_{RI} \) with a string \( s \in L(PG) \cap L(CR)^3 \), where the PG (i.e., \( w \) and \( r \) component of \( s \) is balanced, and the CR (i.e., \( ( \cdot, \cdot) \) component of \( s \) contains exactly the \( \alpha(A - D) \) string of unbalanced returns, followed by the \( \alpha(B - D) \) string of unbalanced calls. (\( D \) is the longest common prefix as expected.) As an example, consider \( x \xrightarrow{w} p \xrightarrow{b} z \xrightarrow{n} w \) in Fig. 5(b); the field component is balanced while the call/ret component reflects that \( x \) flows from the context of the call at line 7 back into main. It is easy to see that if there is a path \( x \sim_{\alpha(\DeltaAB)} y \) and a path \( y \sim_{\alpha(\DeltaBC)} z \), then there is a path \( x \sim_{\alpha(\DeltaAB) \oplus \alpha(\DeltaBC)} z \). By the concatenation lemma, this is exactly \( x \sim_{\alpha(\DeltaAC)} z \).

**Theorem 5.2.** Let \( C, S, \mathbb{H} \) be a program state and let \((x^A, y^B)\) \( \in C \). The following statements are true.

- There is a path \( x \sim_{\alpha(\DeltaAB)} y \) in \( G_{RI} \).
- If \( \alpha(B) > y \) is mutable according to reference immutability (ReIm), then there is an inverse path \( y \sim_{\alpha(\DeltaBA)} x \) in \( G_{RI} \).

A corollary is that if \( y \) is an update, i.e., we have a write \( y.f = z \), then there is an inverse path \( y \sim_{\alpha(\DeltaBA)} x \) in \( G_{RI} \). (By definition ReIm types an update as mutable and therefore \( \alpha(B) > y \) is mutable; the inverse path follows from the second clause of the theorem.) An inverse path is not necessarily made up of inverse edges, it is just the “inverse” of \( x \sim_{\alpha(\DeltaAB)} y \).

**Proof.** The prove is by standard induction and case-by-case analysis of program statements. Given transition \( \llbracket s \rrbracket(A, C, S, \mathbb{H}) = C', S', \mathbb{H}' \) if the lemma holds on all states up to \( C, S, \mathbb{H} \), including

---

3As it is standard, for the purposes of the intersection, we extend the alphabets of grammars PG and CR to include the symbols of the other grammar.
\( \mathbb{C}, \mathbb{S}, \mathbb{H}, \) then it holds on \( \mathbb{C}', \mathbb{S}', \mathbb{H}' \). We note that even though the structure of the proof is standard, the treatment of individual statements is involved.

Case 1. Consider statement \( x = y \). By the inductive hypothesis, there is a path \( \alpha(\Delta AB) \xrightarrow{\sim} y \in G_{RI} \) for every \( (\gamma^A, \gamma^B) \in \mathbb{C} \). Since there is an edge \( y \rightarrow x \in G_{RI} \) (by construction of \( G_{RI} \)), there is a path \( \alpha(\Delta AB) \xrightarrow{\sim} x \in G_{RI} \). To show the second clause of the theorem, assume \( \alpha(B) \triangleright x \) is mutable. Therefore, \( \alpha(B) \triangleright y \) is mutable, and by the inductive hypothesis there is an inverse path \( \gamma \xrightarrow{\sim} v \in G_{RI} \). \( \alpha(B) \triangleright x = \text{mutable implies that } x \text{ is either poly (i.e., there is a }R\)-path to an update), or mutable (an \( M/C\)-path to update). Thus, there is an inverse edge \( x \rightarrow y \in G_{RI} \) by construction. Adding the inverse edge to the inverse path yields \( x \xrightarrow{\sim} v \), as needed (concatenation lemma).

Case 2. Consider call \( i : x = y . m(z) \). By induction, there is path \( \alpha(\Delta AB) \xrightarrow{\sim} z \in G_{RI} \) for each \( (\gamma^A, \gamma^B) \in \mathbb{C} \). The dynamic semantics appends the new fresh frame \( f \) onto \( B \) to get new context \( B' = B \circledast f \), and new chain \( (\gamma^A, \gamma^B') \in \mathbb{C'} \). There is an edge \( z \rightarrow p \) and thus path \( \alpha(\Delta AB') \xrightarrow{\sim} p \in G_{RI} \). Again, the second clause is more involved. Assume \( \alpha(B') \triangleright p \) is mutable. Then \( \alpha(B) \triangleright z \) is mutable as well. (If \( p \) is mutable, then by the rules of \( \text{ReIm} \), \( z \) is mutable, and therefore, \( \alpha(B) \triangleright z \) is mutable. Otherwise, \( p \) is poly and \( \alpha(B') \) is mutable. Since \( \alpha(B') = \alpha(B) \triangleright x \), there are two cases, \( \alpha(B) \) is mutable and \( x \) is poly, or \( x \) is mutable. In the first case, by the rules of \( \text{ReIm} \) we must have \( z \) poly, which yields \( \alpha(B) \triangleright z = \text{mutable} \). If \( x \) is mutable, then by the rules of \( \text{ReIm} \) \( z \) must be mutable, which again yields \( \alpha(B) \triangleright z = \text{mutable} \).) Since we have established that \( \alpha(B) \triangleright z \) is mutable, we have an inverse path \( z \xrightarrow{\sim} v \in G_{RI} \). Since \( \alpha(B') \triangleright p \) is mutable, \( p \) is either mutable or poly. In the case of poly, we must have \( \alpha(B') = \text{mutable} \) which entails that \( x \) is not readily, and therefore, \( x \triangleright p \) is not readonly; thus, the analysis adds an inverse edge \( p \xrightarrow{\sim} z \). Therefore, \( p \xrightarrow{l_i} z \xrightarrow{\sim} v \Rightarrow p \xrightarrow{\sim} z \xrightarrow{\sim} v \Rightarrow p \xrightarrow{\sim} v \) (by Lemma 5.1), which is the expected inverse path.

Case 3, return \( i : x = y . m(z) \) is analogous to Case 2.

Case 4. The most interesting case arises when the runtime semantics adds new chains at a field read. Let \( y = x . f \) in context \( B \) and \( x . \prime f = y \) in \( A \) be such that \( x \) and \( x' \) refer to \( o \) and \( x' . f = y' \) is the most recent write to \( o . f \) preceding the read out of \( x . f \). That is, this is the last write that set \( [o . f \leftarrow ...] \) to form some \( \mathbb{C}'' \). By the inductive hypothesis, for every chain \( (\gamma^D, (\gamma'')^A) \), in this case in the earlier \( \mathbb{C}'' \), there is a path \( \alpha(\Delta AD) \xrightarrow{\sim} y' \in G_{RI} \). By Lemma 3.1 there are chains \( (w^C, (x')^A) \in \mathbb{C}'' \) and \( (w^C, x^B) \in \mathbb{C} \), where \( w = \text{new } o \) in context \( C \) is the allocation site of \( o \). By the inductive hypothesis, we have paths \( w \xrightarrow{\alpha(\Delta AC)} x' \in G_{RI} \), \( w \xrightarrow{\alpha(\Delta CB)} x \in G_{RI} \), as well as an inverse path \( x' \xrightarrow{\sim} w \in G_{RI} \) since \( x' \) is mutable. Thus, we have a path

\[
\begin{align*}
\alpha(\Delta DA) \xrightarrow{\sim} y' \xrightarrow{w} x \xrightarrow{\alpha(\Delta AC)} w \xrightarrow{\alpha(\Delta CB)} x \underset{\text{ri}}{\xrightarrow{\sim}} y \Rightarrow v \xrightarrow{\sim} y \in G_{RI}
\end{align*}
\]

Now consider the second clause of the theorem. Suppose \( \alpha(B) \triangleright y \) is mutable; we need to show inverse path \( y \xrightarrow{\sim} v \in G_{RI} \). If \( \alpha(B) \triangleright y \) is mutable, then \( y \) is mutable or poly and by the rules of \( \text{ReIm} \) \( f \) is poly and \( x' \) is mutable. Therefore \( y' \) is mutable and there is an inverse path \( y' \xrightarrow{\sim} v \in G_{RI} \). Next, if \( \alpha(B) \triangleright y \) is mutable, then \( \alpha(B) \triangleright x \) is mutable and by the inductive hypothesis we have an inverse \( x \xrightarrow{\sim} w \in G_{RI} \). \( y \) being mutable or poly also implies that we have added an inverse edge \( y \xrightarrow{\sim} x \) during construction of \( G_{RI} \). Adding all these paths along with
the original $w \xrightarrow{\alpha(\Delta CA)} x'$ yields
\[
yw \xrightarrow{\alpha(\Delta BC)} wx \xrightarrow{\alpha(\Delta CA)} x' \xrightarrow{\alpha(\Delta AD)} v \Rightarrow y \xrightarrow{\alpha(\Delta BD)} v \in G_{RI}
\]

Of course, even though we have shown that each chain has, roughly speaking, a corresponding path $p$ in the intersection of $L(PG)$ and $L(CR)$, computing the exact set of paths $p$ is undecidable. We define an approximate reachability analysis over $G_{RI}$, CFL, and we show that each path $p$ has a corresponding representative path in CFL. In the following section we define type-based FlowCFL and interpret it in terms of CFL-reachability. In Sect. 7 we define the CFL algorithm and establish equivalence of FlowCFL and CFL thus proving FlowCFL correct.

6 TYPE-BASED ANALYSIS

This section presents the type-based FlowCFL outlining parallels with CFL-reachability as described in Sect. 4. The reader may wonder why one needs a type system, when one has a clear semantics in terms of standard CFL-reachability. First, type systems and type-based taint analysis have already been used in the literature [Huang et al., 2014, 2015, Sampson et al., 2011, Shankar et al., 2001], in some cases without correctness proofs. CFL-reachability brings insight into type-based reachability/taint analysis and the theory of type qualifiers [Foster et al., 2002], and presents a novel framework for reasoning about correctness. Second, a type system allows programmers to specify requirements with type qualifiers, e.g., $\text{pos} x$, and take advantage of systems such as the Checker Framework (https://checkerframework.org/) to statically check these requirements; such requirements cannot be easily expressed or checked using CFL-reachability. Third, type systems are modular, while CFL-reachability systems are typically whole-program analyses. A significant advantage of a type-based interpretation is that it allows for modular reasoning. We can infer type annotations for libraries (e.g., as in [Huang et al., 2012b]), then type check a user program against annotated libraries while handling callbacks via standard function subtyping.

Sect. 6.1 describes the type qualifiers in FlowCFL and Sect. 6.2 describes the typing rules. Sect. 7 establishes equivalence of a certain CFL-reachability analysis and the type-based analysis.

6.1 Type Qualifiers

FlowCFL makes use of the $\text{pos}$ and $\text{neg}$ type qualifiers that we introduced in Sect. 2.1:

- $\text{pos}$ — a $\text{pos}$ variable $x$ is a source or $x$ is such that a source flows to $x$. A $\text{pos}$ $x$ or any of its components cannot flow to a sink. For example
  \[
y = x.f
  \]
  where $y$ is a sink, is not allowed. Similarly,
  \[
y = \text{id}(x); z = y.f;
  \]
  where $z$ is a sink and $\text{id}$ is the identity function, is not allowed.

- $\text{neg}$ — a $\text{neg}$ variable $x$ is a sink, or $x$ flows to a sink.

- $\text{poly}$ — a $\text{poly}$ variable expresses polymorphism. In some contexts, $\text{poly}$ is interpreted as $\text{pos}$ and in other contexts, it is interpreted as $\text{neg}$.

The subtyping hierarchy with $\text{poly}$ becomes

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

It is allowed to assign a $\text{poly}$ variable into a $\text{pos}$ one, but not the other way around; similarly, it is allowed to assign a $\text{neg}$ variable into a $\text{poly}$ one, but not the other way around. Subtyping in
FlowCFL models flow of values which is non-standard. The poly value is interpreted as either neg or pos, depending on the stack context. It would be safe to assign a neg value into a poly variable (which becomes either neg or pos) without causing flow from pos to neg. However, it would not be safe to assign a pos value into a poly variable because it may become neg, causing flow from a pos to a neg variable.

We define the adaptation operation, analogously to the operation in Sect. 4.3.

\[
\begin{align*}
_\top &\triangleright pos = pos \\
_\top &\triangleright neg = neg \\
q &\triangleright poly = q
\end{align*}
\]

Again, a pos or neg variable remains pos or neg. As in Sect. 4.3 a poly variable takes the value of the adapter (i.e., context of adaptation): if the adapter is pos, then poly is interpreted as pos, and if the adapter is neg then poly is interpreted as neg.

To avoid clutter we have used the same notation for the adaptation operator, $\triangleright$, as in Sect. 4.3. From now on, we will use $\triangleright$ to refer to the above definition (FlowCFL), and we will use $\triangleright_{RI}$ to refer to the adaptation operator of reference immutability in the rare occasions it comes into play.

Adaptation adapts fields, formal parameters, and return values according to the context at the field access and method call. The type of a poly field $f$ takes the value of the receiver at the field access. The type of a poly parameter or return is interpreted by adapters at call site $i$. We elaborate on this shortly.

### 6.2 Typing Rules

The typing rules for program statements appear in Fig. 6. The rules are defined in terms of a type environment $\Gamma$, which is standard. $\Gamma = \langle C, \sigma \rangle$, where $C$ is a set of subtyping constraints and $\sigma$ is a map from program variables to type qualifiers: $\sigma: V \rightarrow \{\text{pos}, \text{poly}, \text{neg}\}$. The premise of each rule in Fig. 6 consists of two parts: one part adds constraints to $C$, and the other part, “$C$ holds”, enforces those constraints. Concretely, “$C$ holds” requires (1) that $C$ is closed under the rules in Fig. 7 and (2) that assignment of qualifiers to variables is such that all subtyping constraints in $C$ hold.

Rule ($\textsc{tassign}$) adds constraint $y \leftarrow x$ to $C$, which forbids assignment of a pos or poly reference to a neg one as well as assignment of a pos reference to a poly one. Again, we abuse notation by eliding qualifiers. Strictly, the above constraint should have been written as $q_y \leftarrow q_x$ where $q_y = \Gamma(y)$ and $q_x = \Gamma(x)$; rules are more compact while still clear. If $x$ is not readonly according to reference immutability, ($\textsc{tassign}$) adds the inverse constraint $x \leftarrow y$ as well. In other words, the expected subtyping constraint turns into an equality constraint. This is a well-known issue, typically referred to as the problem of covariant arrays [Bank et al., 1997], which stipulates that subtyping is unsafe in the presence of mutable references. The standard solution, adopted by the majority of systems, e.g., Enerj [Sampson et al., 2011], is to impose equality constraints for all references. This is akin to the bidirectional flow graph in Sect. 4. Our solution combines the “bidirectional” system with reference immutability to achieve limited subtyping and better precision.

One immediately notices the parallel with CFL-reachability. $y \leftarrow x$ corresponds to forward edge $y \xrightarrow{d} x$ in $G_{RI}$ and the inverse constraint $x \leftarrow y$ corresponds to the inverse edge $x \xleftarrow{d} y$. The same reasoning applies to all other explicit and implicit assignments: if the left-hand-side is readonly then the rule enforces a subtyping constraint, otherwise it adds an inverse constraint, thus ensuring an equality just as in Sect. 4.

Rules ($\textsc{twrite}$), ($\textsc{tread}$) and ($\textsc{tcall}$) use adaptation. At field accesses $y.f$, field $f$ is interpreted in the context of receiver $y$. If $f$ is pos (or neg in the positive setting), then its adapted value remains pos (or neg). If $f$ is poly, then the adapted value assumes the type of $y$. Notably, FlowCFL restricts the type of $f$ to $\{\text{pos, poly}\}$ in the negative setting, and to $\{\text{poly, neg}\}$ in the positive setting. In our
Fig. 6. Typing rules associated to program statements. \( \Gamma \vdash A \) means \( C, \sigma \vdash A \) as \( \Gamma = (C, \sigma) \). "C holds" requires (1) that \( C \) is closed under the rules from Fig. 7 and (2) that all constraints in \( C \) type check.

discussion going forward, we assume the negative setting (as described in Sect. 2.1), however all reasoning applies to the symmetric positive setting as well. We can allow neg fields in FlowCFL. However, we are interested in type inference, and allowing neg fields would create ambiguity: if \( x.f \) flows to neg, (1) do we infer that field \( f \) is neg, and is "special", i.e., it is excluded from the state of a potentially pos \( x \), or (2) do we infer that \( f \) is just a "regular" field, and a negative \( x.f \) entails a neg \( x \)? Restricting fields to \{pos, poly\} chooses the latter, as there is no way to know, without programmer annotations, which fields are "special". Inference tools such as Javafier [Quinonez et al., 2008] and RelmInfer [Huang et al., 2012b] make the same choice. The restriction states that a positive reference \( x \) cannot have negative components.

Rules (twrite) and (tread) handle heap-transmitted dependences. Consider

\[
x.f = a; \quad y = x; \quad \text{neg} \ b = y.f
\]

Rules (twrite), (assign), and (tread) create constraints

\[
a <: x \triangleright f \quad x <: y \quad y \triangleright f <: b
\]

Since \( b = \text{neg} \), \( f \) is poly and we have

\[
a <: x \quad x <: y \quad y <: b
\]

which forces \( a = \text{neg} \), as needed. Notice again the parallel with CFL-reachability. Constraints

\[
a <: x \triangleright f \quad x <: y \quad y \triangleright f <: b
\]

correspond to the path in \( G_{RI} \)

\[
a \xrightarrow{\text{wq}} x \xrightarrow{d} y \xrightarrow{r_q} b
\]
Fig. 7. Constraint propagation. p and ret in (TRANS-CALL) can be any of this, p, or ret.

and the "linear" constraints

\[ a \ll x \quad x \ll y \quad y \ll b \]

correspond to dropping the \( w_f \) and \( r_f \) annotations, thus achieving a CSFI approximation. FlowCFL is a more precise variant of CSFI as we argue in Sect. 7.

Rule (TCALL) captures call-transmitted dependences and is the most involved. Unlike previous systems, e.g., EnerJ and DroidInfer, FlowCFL allows for distinct adapters. Every parameter/return has a distinct associated adapter \( q^i_{\text{this}} \), \( q^j_p \) and \( q^j_{\text{ret}} \) instead of a single per-call-site adapter \( q^i \). This is necessary to achieve the CFL-reachability semantics. We elaborate on this shortly.

Before we delve into (TCALL), we consider the rules in Fig. 7. The rules explicitly collect transitive intraprocedural constraints into \( C \); they capture constraints that correspond to call/ret balanced paths (i.e., M-paths). (ERASE-LEFT) and (ERASE-RIGHT) "linearize" constraints—e.g., when \( f \) is poly, \( y \ll f <: b \) becomes \( y \ll b \). This corresponds to dropping field \( w_f \), \( r_f \) annotations, thus achieving a variant of CSFI. Notably, a constraint is linearized only if the corresponding field is poly, which happens only when the field is on a path to a sink.

Rule (TRANS-CALL) in Fig. 7 transfers constraints from the callee to the caller. If there is flow from a parameter \( p \) to return \( ret \), captured by subtyping constraint \( p <: ret \in C \), then there is flow from actual argument \( z \) to the left-hand-side of the call assignment \( x \).

Consider the example below, which is similar to Fig. 2:

```java
1  class A {
2     poly B f;
3     void set(poly A this, poly B p) {
4         this.f = p;
5     }
6     poly B get(poly A this) {
7         ret = this.f;
8         return ret;
9     }
10 }
```

```java
1  main() {
2      neg A e = new A;  \( q \)
3      B a = new B;
4      e.set(a); // \( q^4_{\text{this}} = q^4_p = \text{neg} \)
5      neg A g = e;
6      neg B b = g.get(); // \( q^6_{\text{this}} = q^6_{\text{ret}} = \text{neg} \)
7  }
```
Class A is polymorphic and main uses A in a negative context. As b is a sink, it follows that a flows to a sink and the types should properly reflect the flow. Line 4 in A.set results in constraint p <: this >> f. Since f is poly, (ebase-right) in Fig. 7 produces p <: this. Call site 4 in main entails

\[ a <: q^4_p > p \quad e <: q^4_{\text{this}} > \text{this} \quad q^4_{\text{this}} > \text{this} \quad e <: e \]

The last constraint is the inverse of the previous one due to the mutation of this. Rule (trans-call) in Fig. 7 combines constraints

\[ a <: q^4_p > p \quad p <: \text{this} \quad q^4_{\text{this}} > \text{this} \quad e <: e \]

to get a <: e. Analogously, constraint this >> f <: ret in get and call site 6 in main yield g <: b. Constraints a <: e, e <: g (due to g = e) and g <: b capture the flow from a to b.

### 6.3 FlowCFL−

Why not use the following simpler (tcall)?

\[
\text{typeof}(m) = \text{this}, p \rightarrow \text{ret} \\
\Gamma \vdash y <: q^l > \text{this} \quad \Gamma \vdash x >_{\text{RI}} \text{this} \quad \text{this is not readonly} \implies q^l > \text{this} \quad \Gamma \vdash y \\
\Gamma \vdash z <: q^l > p \quad \Gamma \vdash x >_{\text{RI}} p \quad \text{p is not readonly} \implies q^l > p <: z \\
\Gamma \vdash q^l > \text{ret} <: x \quad \Gamma \vdash x \quad \text{is not readonly} \implies x <: q^l > \text{ret} \\
\Gamma \vdash i : x = Y.m(z)
\]

The rule makes use of a single viewpoint adapter q^l rendering C and the rules in Fig. 7 unnecessary! Replacing (tcall) in Fig. 6 with the above (tcall) yields a new type system, which we call FlowCFL− (FlowCFL minus). An advantage of FlowCFL− is its simplicity; it is also sound, however, it rejects programs that CFL-reachability over G_{RI} handles precisely.

The following (somewhat contrived) example illustrates the imprecision of FlowCFL− and the need for multiple adapters:

```
1 poly Y m(poly A this, poly X p, poly Y q) {
2   this.f = p;
3   ret = q;
4 }
5 ...
6 A a, X x, Y y;
7 Y y2 = a.m(x,y);
8 neg X x2 = a.f;
9 A a1, X x1, Y y1;
10 neg Y y3 = a1.m(x1,y1);
```

There are two disjoint paths through m: (1) p ↼ this, and (2) q ↼ ret. At call 7 the first path appears in negative context (as subpath of the path from x to neg x2), while the second path appears in positive context (as subpath of the path from y to y2). At call 11 the opposite happens: the first path appears in positive context and the second one in negative context. CFL-reachability precisely propagates the negative qualifier neg x2 back to x and neg y3 back to y1. FlowCFL propagates the negative qualifiers in exactly the same way. It discovers paths x ↼ x2 and y1 ↼ y3 via C: it adds x <: x2 and y1 <: y3 to C based on p <: this and q <: ret respectively; it adds no spurious constraints (i.e., paths).
In contrast, with a single adapter $q^i$ (e.g., as in DroidInfer and EnerJ) the above precise typing is impossible. This is because the role of the adapter is twofold: (1) to interpret the poly parameter/return in the corresponding context, and (2) to propagate paths from callee to caller. Given sink neg X x2 in line 8, field f is poly (due to the flow of f to sink x2). This forces this and p of m to poly, as shown in the typing of m in lines 1-4. Sink neg Y y2 in line 11 forces ret and q to poly as well. Due to $q^i$ >> this: a and $q^{1i}$ >> ret: y3 respectively, we have $q^i = \text{neg}$ and $q^{1i} = \text{neg}$. Thus, $y < q^i >> q$ and $a1 << q^{1i} >> this$ unnecessarily force y and a1 to neg.

Multiple adapters differentiate between flow paths. This is because the purpose of the adapters is only to interpret the poly parameter/return in the corresponding context; propagation of paths from callee to caller is done via $C$. In our example we have $a1 << q^{1i} >> this$, $x1 << q^{1i} >> p$, $y1 << q^{1i} >> q$, and $q^{1i} >> ret$: y3, $q^{1i}_{\text{this}} = q^{1i} = \text{pos}$, and $q^{1i}_{\text{ret}} = q^{1i} = \text{neg}$. The qualifiers are flipped at call site 7: $q^{2i}_{\text{this}} = q^{2i} = \text{neg}$, and $q^{2i}_{\text{ret}} = q^{2i} = \text{pos}$.

## 7 EQUIVALENCE OF CFL-REACHABILITY AND TYPE-BASED ANALYSES

Recall that CFL-reachability over both w, r and call/ret annotations is undecidable. Typical approximations are CSFI (context-sensitive, field-insensitive) and CIFS, and variants in-between. FlowCFL captures a variant of CSFI, which we call CSFI+. As mentioned earlier, CSFI replaces all field annotations with d and performs CR-reachability. E.g., in

$$x.f = a; \text{ neg } b = x.g; \implies a \xrightarrow{d} x \xrightarrow{d} b$$

CSFI replaces $w_f$ and $r_g$ with d and spuriously propagates neg b back to a. Another way to look at CSFI is as if we replaced production $B \rightarrow w_f B r_f$ in the PG grammar in Fig. 4(b) with $B \rightarrow w_f B r_g$.

Like CSFI, CSFI+ does match certain distinct field annotations $w_f$ and $r_g$, but not all. CSFI+ matches $w_f$ and $r_g$ only if fields f and g both flow to sinks. As an example of potential imprecision in CSFI+ consider the two snippets of the same program:

1. $x.f = a0;$
2. $x.g = b0;$
3. $\text{neg } c0 = x.f;$
4. $d0 = x.g;$

$$
\begin{align*}
\text{a0} & \xrightarrow{w_f} \text{x} \xrightarrow{r_f} \text{c0} \\
\text{b0} & \xrightarrow{w_g} \text{d0}
\end{align*}
$$

1. $y.f = a1;$
2. $y.g = b1;$
3. $c1 = y.f;$
4. $\text{neg d1 = y.g;}

$$
\begin{align*}
\text{a1} & \xrightarrow{w_f} \text{y} \xrightarrow{r_f} \text{c1} \\
\text{b1} & \xrightarrow{w_g} \text{d1}
\end{align*}
$$

Since both f and g flow to sinks, CSFI+ matches the distinct field annotations. It propagates negative c0 to both a0 and b0. Similarly, it propagates negative d1 to both a1 and b1.

The problem is to find a set of paths that includes all properly matched w/r and call/ret paths to sinks in $G_RI$. Without loss of generality we assume that sinks are primitive types, i.e., the PG-component of every path from v to a sink is either a G-path or a B-path. CSFI+ back-propagates sinks maintaining a set $F$ of fields that flow to sinks. The difference between precise propagation, CSFI+, and CSFI lies in production $B \rightarrow w_f B r_g$. Precise propagation infers a B-path when $f = g$ (as in Fig. 4(b)), CSFI+ infers a B-path only when $\{f, g\} \in F$, and CSFI infers a B-path in all cases.

Fig. 8 presents two equivalent implementations of CSFI+. Both algorithms implement FlowCFL in the negative setting—they start from a set of sinks and back-propagate those sinks via CSFI+-reachability. Algorithm CFl collects all paths from variables to sinks in $P$, as well as all balanced subpaths of these paths (M-paths). One can easily show (by induction on the length of the path) that CFl captures in $P$ all properly matched paths in $G_RI$. Algorithm TYPES makes use of the type system in Sect. 6. It computes a map $S$ from program variables to sets of qualifiers. $S$ is initialized as follows: $S(u) = \{\text{neg}\}$ for each sink $u$, $S(x) = \{\text{pos, poly, neg}\}$ for each variables $x$, and $S(f) = \{\text{pos, poly}\}$ for each field $f$. TYPES iterates through program statements; it infers new "linear" constraints and
removes infeasible qualifiers from variable sets until it reaches a fixed point. Function \textit{Solve} takes a constraint, e.g., \( x \prec: y \) and updates \( S(x) \). E.g., if \( S(y) = \{ \text{neg} \} \) and \( S(x) = \{ \text{pos, poly, neg} \} \), \textit{Solve} removes \text{pos} and \text{poly} from \( S(x) \) because neither is a subtype of \text{neg}. As another example, consider constraint \( x \succ f \prec: y \) where \( S(y) = \{ \text{poly, neg} \} \), \( S(x) = \{ \text{pos, poly, neg} \} \), and \( S(f) = \{ \text{pos, poly} \} \). \textit{Solve} removes \text{pos} from both \( S(x) \) and \( S(f) \) because the constraint cannot be satisfied if either \( x \) or \( f \) is pos. Such fixpoint iteration has been used in previous work [Huang et al., 2012a, Kiezun et al., 2007, Tip et al., 2011].
TYPES assigns sets of qualifiers to variables. To assign a final typing to a variable/field, we pick the maximal qualifier according to preference ranking pos > poly > neg. One can see through case analysis by case by case analysis that the maximal typing type checks with the rules in Fig. 6 and Fig. 7. Qualifiers \( q_i \), \( q_p \), \( q_{et} \) can take any value that satisfies the maximal typing.

We argue correctness of our type-based analysis by establishing equivalence between CFL and Types. Def. 7.1 states that if there is an M/C-path or an R-path from \( x \) to a sink \( n \), then the maximal type of \( x \) in \( S \) is at least, respectively, neg or poly. For example, if there is an R-path, the maximal typing is poly or neg.

**Definition 7.1.** (Soundness) \( P \Rightarrow S \) if and only if

1. \( x \overset{M/C}{\sim} n \in P \Rightarrow \max(S(x)) <: \text{neg} \)
2. \( x \overset{R}{\sim} n \in P \Rightarrow \max(S(x)) <: \text{poly} \)

Def. 7.2 states that \( x \)'s maximal type in \( S \) implies a corresponding path in \( P \). For example, maximal typing poly means that there is a R-path but there is no M/C-path.

**Definition 7.2.** (Precision) \( S \Rightarrow P \) if and only if

1. \( \max(S(x)) = \text{neg} \Rightarrow \exists x \overset{M/C}{\sim} n \in P \)
2. \( \max(S(x)) = \text{poly} \Rightarrow \exists x \overset{R}{\sim} n \in P \land \exists x \overset{M/C}{\sim} n \in P \)
3. \( \max(S(x)) = \text{pos} \Rightarrow \text{no path from } x \text{ to any } n \in P \)

**Definition 7.3.** (Equivalence) \( P \equiv S \) if and only if \( P \Rightarrow S \) and \( S \Rightarrow P \).

Let the Hoare triple denote parallel execution of Edge and Constraint on statement \( s \):

\[ \{P, S\} \Edge(e(s)) \parallel \Constraint(c(s)) \{P', S'\} \]

The equivalence result comes from the following theorem:

**Theorem 7.4.** If \( P \equiv S \) and \( \{P, S\} \Edge(e(s)) \parallel \Constraint(c(s)) \{P', S'\} \) then \( P' \equiv S' \).

**Proof.** The proof is carried out by case-by-case analysis as in [Milanova, 2018]. \( \square \)

8 RELATED WORK

CFL-reachability dates decades back [Reps, 2000, Reps et al., 1995], yet it remains highly relevant. Zhang and Su [Zhang and Su, 2017], Spath et al. [Späth et al., 2019], and Chatterjee et al. [Chatterjee et al., 2018], among other works, present novel CFL-reachability approximations and algorithms with application to data dependence. Xu et al. [Xu et al., 2009], and Lu and Xue [Lu and Xue, 2019], again among other works, present novel CFL-reachability-based points-to analyses. In all works, the concept of the inverse edge, introduced by Sridharan et al. [Sridharan et al., 2005], factors in. Our work presents a formal treatment of the inverse edges and paths and a correctness argument for CFL-reachability over graphs with inverse edges. Recent work by Li et al. [Li et al., 2020] presents a graph simplification algorithm for CFL-reachability that removes certain edges that do not contribute to paths to sinks. This work nicely complements our work, as it can be applied on any CFL-reachability graph, including \( G_{BI} \) and \( G_{BL} \). Li et al., demonstrate their technique using DroidInfer’s graphs [Huang et al., 2015], which are \( G_{BI} \) graphs. (We use DroidInfer’s graphs in our experiments as well.) We have focused on understanding the dynamic semantics of flows, establishing soundness of the removal of certain inverse edges, and drawing a connection between CFL-reachability and type-based flow analysis.
Type-based analysis has a long history as well [Palsberg, 2001] and our analysis falls into this line of work. Classical work on type-based taint (information flow) analysis includes work by Shankar et al. [Shankar et al., 2001], Volpano et al. [Volpano et al., 1996], and Myers [Myers, 1999].

Few works have explored the connection between CFL-reachability and type-based analysis. Milanova [Milanova, 2018] presents an interpretation of reference immutability in terms of CFL-reachability. We make use of this interpretation (Sect. 4.3), however, we address a different and more difficult problem, as the nature of approximation in reference immutability [Huang et al., 2012b, Milanova, 2018, Tschantz and Ernst, 2005] renders inverse edges unnecessary and reachability analysis much simpler.

Rehof and Fahndrich [Rehof and Fähndrich, 2001] connect type-based flow analysis and CFL-reachability. However, Rehof and Fahndrich do not discuss mutable references and it is unclear how their analysis and interpretation, targeting a pure functional language, can handle mutable data and heap-transmitted dependences. On the other hand, Rehof and Fahndrich handle higher-order functions while we do not. An important direction of future work is extending our approach with handling of higher-order functions which will enable application of the FlowCFL framework to the analysis of dynamic languages. Fahndrich et al. [Fähndrich et al., 2000] apply the theory of [Rehof and Fähndrich, 2001] to build a context-sensitive Steensgaard-style points-to analysis for C, thus using equality constraints instead of subtyping constraints. As mentioned earlier, equality constraints is the standard approach to the handling of mutable references [Fuhrer et al., 2005, Sampson et al., 2011, Shankar et al., 2001].

9 CONCLUSION

We presented FlowCFL, a framework for type-based reachability analysis. We presented (1) a novel dynamic semantics, (2) correctness arguments for CFL-reachability over graphs with inverse edges, and (3) equivalence between a CFL-reachability analysis and a type-based reachability analysis.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, pages 259–269, 2014.

Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’97, pages 132–145, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263714. URL http://doi.acm.org/10.1145/263699.263714.

James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain: a first-order type for uncertain data. In Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14, Salt Lake City, UT, USA, March 1-5, 2014, pages 51–66, 2014.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 33–52, 2013a.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware (appendix). http://groups.csail.mit.edu/pac/rely/rely13appendix.pdf, 2013b.

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. PACMPL, 2(POPL):1–30:30, 2018. doi: 10.1145/3158118. URL https://doi.org/10.1145/3158118.

Michael Cooney. IBM touts encryption innovation: New technology performs calculations on encrypted data without decrypting it. Network World, June 2009.

Werner Diehl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings, pages 333–357, 2011. doi: 10.1007/978-3-642-22655-7_16. URL https://doi.org/10.1007/978-3-642-22655-7_16.

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. ACM Transactions on Programming Languages and Systems, 34(1):1–48, April 2012. ISSN 01640925.
Yao Dong, Ana Milanova, and Julian Dolby. Jcrypt: Towards computation over encrypted data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 8:1–8:12, 2016.

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Rosner, Karl Koscher, Paulo Barros, Ravi Bhorskar, Seunghyeop Han, Paul Vines, and Edward Xuejun Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1092–1104, 2014.

David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. NOW Publishers, Boston-Delft, 2018.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI ’00*, pages 253–263, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349332. URL http://doi.acm.org/10.1145/349299.349332.

Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 32nd European Conference on Object-Oriented Programming, ECOOP 2018, June 03–05, 2018, Woodstock, NY*, Milanova, Ana

Craig Gentry. Computing arbitrary functions of encrypted data. *JACM, 53(3):97–105, March 2010*. ISSN 0001-0782.

Craig Gentry and Shai Halevi. Implementing Gentry’s fully-homomorphic encryption scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology, EUROCRYPT ’11*, pages 129–148, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20464-7.

Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Osokin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 279–293, 2016.

Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP’12*, pages 181–206, Berlin, Heidelberg, 2012a. Springer-Verlag. ISBN 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_9. URL http://doi.acm.org/10.1145/978-3-642-31057-7_9.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfe: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’12*, pages 879–896, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384680. URL http://doi.acm.org/10.1145/2384616.2384680.

Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for java web applications. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014*, Proceedings, pages 140–154, 2014. doi: 10.1007/978-3-642-54804-8_10. URL https://doi.acm.org/10.1145/978-3-642-54804-8_10.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 15th International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 106–117, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771803. URL http://doi.acm.org/10.1145/2771783.2771803.

Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing java classes. In *Proceedings of the 29th International Conference on Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014*, Proceedings, pages 140–154, 2014. doi: 10.1007/978-3-642-54804-8_10. URL https://doi.acm.org/10.1145/978-3-642-54804-8_10.

Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved Dyck-Reachability. In *Proceedings of the 41th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, ACM. doi: 10.1145/3385412.3386021. URL https://doi.acm.org/10.1145/3385412.3386021.

Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *PACMPL, 3(OOPSLA):148:1–148:29, 2019*. doi: 10.1145/3360657. URL https://doi.acm.org/10.1145/3360657.

Ana Milanova. Definite reference mutability. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 25:1–25:30, 2018. doi: 10.4230/LIPIcs.ECOOP.2018.25. URL https://doi.org/10.4230/LIPIcs.ECOOP.2018.25.
Ana Milanova and Wei Huang. Composing polymorphic information-flow systems with reference immutability. In Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013, pages 5:1–5:7, 2013. doi: 10.1145/2489804.2489809. URL https://doi.org/10.1145/2489804.2489809.

Andrew C. Myers. Jflow: Practical mostly-static information flow control. In POPL ’99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999, pages 228–241, 1999. doi: 10.1145/292540.292561. URL https://doi.org/10.1145/292540.292561.

Jens Palsberg. Type-based analysis and applications. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE’01, Snowbird, Utah, USA, June 18-19, 2001, pages 20–27, 2001. doi: 10.1145/379605.379635. URL https://doi.org/10.1145/379605.379635.

Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In Proceedings of the 22Nd European Conference on Object-Oriented Programming, ECOOP ’08, pages 616–641, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_26. URL http://dx.doi.org/10.1007/978-3-540-70592-5_26.

Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. Inf. Softw. Technol., 40(11-12):701–726, 1998. doi: 10.1016/S0950-5849(98)00093-7. URL https://doi.org/10.1016/S0950-5849(98)00093-7.

Thomas W. Reps. Program analysis via graph reachability. Inf. Softw. Technol., 22(1):162–186, 2000. doi: 10.1109/345099.345137. URL https://doi.org/10.1109/345099.345137.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Conference Record of POPL ’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995, pages 49–61, 1995. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

Adrian Sampson, Werner Diell, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 164–174, 2011.

Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. Practical secure computation outsourcing: A survey. ACM Comput. Surv., 51(2):31:1–31:40, 2018. doi: 10.1145/3158363. URL https://doi.org/10.1145/3158363.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10, SYM’01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267612.1267628.

Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. PACMPL, 3(POPL):48:1–48:29, 2019. doi: 10.1145/3290361. URL https://doi.org/10.1145/3290361.

Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for java. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’06, pages 387–400, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134027. URL http://doi.acm.org/10.1145/1133981.1134027.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, pages 59–76, 2005. doi: 10.1145/1094811.1094817. URL https://doi.org/10.1145/1094811.1094817.

Sai Deep Tetali. Program Analyses for Cloud Computations. PhD thesis, University of California, 2015.

Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. ACM Transactions on Programming Languages and Systems, 33(3):1–47, April 2011. ISSN 0164-0925. doi: 10.1145/1961204.1961205.

Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA ’05, pages 211–230, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094828. URL http://doi.acm.org/10.1145/1094811.1094828.

Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP’10, pages 304–328, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1884000.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. Journal of Computer Security, pages 167–187, 1996.
public class IntPair {
  int x;
  int y;
  int numAdditions = 0;
  void addToBoth(IntPair this; int amount) {
    x += amount;
    y += amount;
    numAdditions++;
  }
}

public class Example {
  public static void main() {
    IntPair i = new IntPair();
    i.addToBoth(10);
    ...
    IntPair j = new IntPair();
    j.addToBoth(k);
    @Precise z = j.x + j.y;
  }
}

Fig. 9. IntPair from EnerJ [Sampson et al., 2011]. Variable z at line 8 in main is precise (@Precise maps to neg in FlowCFL), and therefore, flow from approximate data to z is forbidden. FlowCFL infers that class IntPair is polymorphic: x, y, this and amount of addToBoth are poly (exactly as annotated in [Sampson et al., 2011]). FlowCFL infers that i in main is @Approx, while j and k are @Precise; i.e., it instantiates polymorphic IntPair as @Approx in the context of i, and as @Precise in the context of j. Field numAdditions in IntPair is @Approx because it does not flow to z in either context (again, exactly as in [Sampson et al., 2011]).

Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_6. URL http://dx.doi.org/10.1007/978-3-642-03013-0_6.

Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pages 344–358, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009848. URL http://doi.acm.org/10.1145/3009837.3009848.

A APPLICATIONS OF FLOWCFL

A.1 Approximate Computing: EnerJ and Rely

In addition to taint analysis, another application domain of FlowCFL is approximate computing, which has received significant attention [Bornholt et al., 2014, Carbin et al., 2013a, Holt et al., 2016, Sampson et al., 2011]. Approximate computing relies on programming language technology such as type systems and Hoare logic to reason about execution on unreliable hardware [Carbin et al., 2013a, Sampson et al., 2011], execution in the presence of probabilistic sensor data [Bornholt et al., 2014], and execution on inconsistent and approximate cloud storage systems [Holt et al., 2016]. An overarching issue is the separation of non-approximate and approximate parts of the program. Currently, all works require large number of manual annotations that explicitly separate the approximate variables and operations from the non-approximate ones.

A.1.1 EnerJ. EnerJ [Sampson et al., 2011] partitions the program variables into @Approx and @Precise where @Approx variables can be stored and used in energy-efficient storage. It requires non-interference for correctness: an @Approx variable cannot flow into a @Precise one. The EnerJ type system can be cast as an instance of FlowCFL in the negative setting. @Approx maps to pos, @Precise maps to neg, and @Context maps to poly. Programmers can annotate a set of @Precise sinks designating values that must be computed precisely. The system infers types for the rest of the variables maximizing the approximate part of the program. Fig. 9 illustrates.
class Newton {
    static float tolerance = 0.00001;
    static int maxsteps = 40;
    static float F(float x) { ... }
    static float dF(float x) { ... }

    static float newton(urel float xs) {
        float x, xprim;
        float t1, t2;
        int count = 0;

        x = xs;
        xprim = xs + 2*tolerance;
        while ((x - xprim >= tolerance) || (x - xprim <= -tolerance)) {
            xprim = x;
            t1 = F(x);
            t2 = dF(x);
            x = x - t1 / t2;
            if (count++ > maxsteps) break;
        }
        if (!((x - xprim <= tolerance) && (x - xprim >= -tolerance))) {
            x = INFTY;
        }
        return x;
    }
}

Fig. 10. Newton’s method from [Carbin et al., 2013a]. Input xs in line 7 is annotated unreliable (urel corresponds to pos in FlowCFL). FlowCFL fills in the remaining annotations. It infers that F is poly float F(poly float x) and so is dF. Variables x, xprim, t1 and t2 are inferred unreliable (as explicitly annotated in [Carbin et al., 2013a]). All operations, except for line 19, are unreliable (as in [Carbin et al., 2013a]).

A.1.2 Rely. Another system in this domain, Rely [Carbin et al., 2013a], reasons about execution on unreliable hardware. Again, programmers must explicitly annotate all unreliable variables (using the urel annotation), as well as all operations on unreliable variables (e.g., unreliable + becomes +). Unannotated variables and operations are considered reliable. [Carbin et al., 2013a] describe how Rely verifies a bound on the reliability of a computation with respect to the reliability of its input. For example, it verifies that the result of the computation in Fig. 10 is at least 0.99 * R(xs), where R(xs) is the reliability of input xs.

Rely can be cast as an instance of FlowCFL in the positive setting. The urel (unreliable) Rely annotation maps to pos, and the default reliable annotation maps to neg. Programmers annotate unreliable inputs with pos and FlowCFL infers types for the rest of the program, thus minimizing the unreliable partition. Unlike with EnerJ where we maximize the approximate partition and thus, energy savings, here we minimize the unreliable partition, which may improve on Rely’s bound. (The smaller the unreliable partition, the more precise the bound on the computation.) Fig. 10 illustrates type inference for Rely. We have run all programs from [Carbin et al., 2013a,b] through FlowCFL and we have inferred the same types as annotated in [Carbin et al., 2013a,b].
public class Data {
    int d;
    void set(Data this, int p) {
        this.d = p;
    }
    int get(Data this) {
        return this.d;
    }
}

public class Example {
    public void main() {
        Data ds = new Data();
        sensitive int s = ...; // sensitive source
        ds.set(s);
        int ss = ds.get();
        Data dc = new Data();
        int c = ...;
        dc.set(c);
        int cc = dc.get();
    }
}

Fig. 11. An example from JCrypt [Dong et al., 2016]. s in line 4 of main is a sensitive input (pos in FlowCFL), and all computation affected by s must be secure. FlowCFL infers that class Data is polymorphic, and that ds and ss in main are sensitive. The remaining variables remain plaintext (neg in FlowCFL).

A.2 Secure Computation

Yet another application of FlowCFL is secure computation. As clients increasingly outsource computation to untrusted cloud servers, there is pressing need to preserve confidentiality of data. This can be done through computation outsourcing [Shan et al., 2018] or Multi-party Computation (MPC) [Evans et al., 2018]. Unfortunately, secure computation is expensive. Fully homomorphic encryption [Cooney, 2009, Gentry, 2010, Gentry and Halevi, 2011] is still prohibitively expensive. Partially homomorphic encryption, an essential building block in both computation outsourcing and MPC, is still costly; for example, homomorphic addition over ciphertexts is about 5X more expensive than addition over plaintexts [Tetali, 2015]. Therefore, it is important to minimize portions of the program that require computation under secure computation protocols.

JCrypt [Dong et al., 2016] is a system where programmers annotate a set of inputs as sensitive (i.e., pos) and JCrypt propagates those sensitive annotations throughout the program. For example, inputs from files in MapReduce applications, or secret-shared inputs in MPC will be annotated as sensitive. JCrypt is yet another instance of FlowCFL in the positive setting; it minimizes the sensitive portion of the program, thus minimizing expensive secure computation. Fig. 11 illustrates JCrypt and inference with FlowCFL.