Boosting the Performance of Alias-Aware IFDS Analysis with CFL-Based Environment Transformers

HAOFENG LI, SKLP, Institute of Computing Technology, CAS, China
CHENGHANG SHI*, SKLP, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
JIE LU, SKLP, Institute of Computing Technology, CAS, China
LIAN LI*, SKLP, Institute of Computing Technology, CAS, China, University of Chinese Academy of Sciences, China, and Zhongguancun Laboratory, China
JINGLING XUE, University of New South Wales, Australia

The IFDS algorithm is pivotal in solving field-sensitive data-flow problems. However, its conventional use of access paths for field sensitivity leads to the generation of a large number of data-flow facts. This causes scalability challenges in larger programs, limiting its practical application in extensive codebases. In response, we propose a new field-sensitive technique that reinterprets the generation of access paths as a Context-Free Language (CFL) for field-sensitivity and formulates it as an IDE problem. This approach significantly reduces the number of data-flow facts generated and handled during the analysis, which is a major factor in performance degradation.

To demonstrate the effectiveness of this approach, we developed a taint analysis tool, IDEDroid, in the IFDS/IDE framework. IDEDroid outperforms FlowDroid, an established IFDS-based taint analysis tool, in the analysis of 24 major Android apps while improving its precision (guaranteed theoretically). The speed improvement ranges from 2.1× to $2,368.4\times$, averaging at 222.0×, with precision gains reaching up to 20.0% (in terms of false positives reduced). This performance indicates that IDEDroid is substantially more effective in detecting information-flow leaks, making it a potentially superior tool for mobile app vetting in the market.

CCS Concepts: • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: IFDS, IDE, CFL-Reachability, Taint Analysis, Alias Analysis

ACM Reference Format:
Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Jingling Xue. 2024. Boosting the Performance of Alias-Aware IFDS Analysis with CFL-Based Environment Transformers. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 364 (October 2024), 29 pages. https://doi.org/10.1145/3689804

1 Introduction

The IFDS framework, initially proposed by Reps et al. [Reps et al. 1995], offers a precise solution for Inter-procedural Finite Distributive Subset (IFDS) data-flow problems. It extends a program's

*Corresponding author

Authors' Contact Information: Haofeng Li, SKLP, Institute of Computing Technology, CAS, Beijing, China, lihaofeng@ict.ac.cn; Chenghang Shi, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, shichenghang21s@ict.ac.cn; Jie Lu, SKLP, Institute of Computing Technology, CAS, Beijing, China, lujie@ict.ac.cn; Lian Li, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China and Zhongguancun Laboratory, Beijing, China, lianli@ict.ac.cn; Jingling Xue, University of New South Wales, Sydney, Australia, j.xue@unsw.edu.au.

ACM 2475-1421/2024/10-ART364
https://doi.org/10.1145/3689804

inter-procedural control flow graph (ICFG) into an exploded graph, with nodes at each program point representing a set of data-flow facts (D), and edges representing data-flow transfer functions from 2^D to 2^D. This allows for context- and flow-sensitive inter-procedural data-flow analysis by identifying inter-procedurally realizable paths in the exploded graph. The IFDS algorithm's time complexity is $O(|E||D|^3)$ and space complexity is $O(|E||D|^2)$, with $|E|$ being the ICFG's edge count.

This framework has been integrated into several analysis tools like Soot [Lam et al. 2011], WALA [IBM 2006], and LLVM [Lattner and Adve 2004], and is used in diverse applications including pointer analysis [Späth et al. 2017; Späth et al. 2016], taint analysis [Arzt et al. 2014; He et al. 2019; Li et al. 2024, 2021; Tripp et al. 2009; Wei et al. 2014], security analysis [Hallem et al. 2002; Manevich et al. 2004; Wassermann and Su 2008], and shape analysis [Gotsman et al. 2006].

Many IFDS problems are *alias-aware*, requiring the simultaneous management of data-flow propagation and aliasing to accurately track data flow through the heap, thus achieving high precision. The original IFDS framework proposed by Reps et al. [1995] does not inherently address aliasing; it relies on an independent pointer analysis [Andersen 1994] to precompute alias information. In order for the analysis results to be precise, it is crucial to use a pointer analysis algorithm that is both fully context- and flow-sensitive. However, achieving this level of sensitivity is still considered overly costly for object-oriented programs [Späth et al. 2016].

In practice, alias discovery within the IFDS framework often involves integrating an on-demand computation of *access paths* [Arzt et al. 2014; He et al. 2019; Späth et al. 2017]. These access paths, abstracted for field-sensitivity, are denoted as $v.f^*$, where v represents a base variable and f^* a sequence of zero or more fields. They are computed as needed and explicitly encoded as data-flow facts to effectively capture tainted aliases. For example, FlowDroid, a prominent IFDS-based taint analysis tool [Arzt et al. 2014], utilizes a forward IFDS solver to propagate tainted access paths, i.e., data-flow facts along control flow. Furthermore, FlowDroid uses a backward solver to compute aliases for newly found tainted access paths or facts at store operations. This approach of using access paths for field-sensitivity enables FlowDroid to perform context- and flow-sensitive analysis, thereby enhancing its analysis precision.

FlowDroid's access-path-based approach, while effective for field-sensitivity, significantly increases the number of data-flow facts and expands the data-flow fact domain D, causing considerable performance overhead. With 5-limited access paths (default in FlowDroid), the number of access paths can be almost an order of magnitude larger than the number of base variables for some programs (Figure 10). Given that the IFDS algorithm [Naeem et al. 2010] scales with cubic time and quadratic space complexity relative to $|D|$, this surge in access paths can severely slow down IFDS-based analysis, making it both compute- and memory-intensive. This limitation, also highlighted in prior studies [Avdiienko et al. 2015; He et al. 2023], limits FlowDroid's scalability, often leading to time-outs and out-of-memory errors in analyzing large Android apps.

An effective strategy for optimizing performance in IFDS problems is to minimize the size of the data-flow fact domain D. One simple approach is to apply k-limiting to access paths, using small values (e.g., 1 or 2) for k. While this can reduce D, it often leads to a significant loss in precision. Additionally, shorter access paths can generate an excessive number of spurious aliases, potentially decreasing the overall performance of IFDS-based analysis. This presents a critical challenge: How can we optimize the domain size of D without compromising precision? This question underscores the need for a balance between efficiency and precision in IFDS problem-solving.

To tackle the challenge of optimizing alias-aware IFDS-based analysis, we propose a new approach that enhances scalability and precision by propagating only the base variables of access paths, thereby effectively reducing the domain size $|D|$. The key novelty lies in treating the generation of access paths as a CFL (Context-Free Language) for field-sensitivity and formulating it as an IDE (Inter-procedural Distributive Environment) problem. Our approach symbolically models k-limited access paths as edge functions in an IFDS/IDE framework, an extension of IFDS with environment transformers [Sagiv et al. 1996]. Field accesses are encoded as edge labels on the program's exploded graph, where only base variables are propagated, and access paths are determined by solving a CFL-reachability problem [Reps 1998; Yannakakis 1990] on strings representing the sequence of accessed fields. Our approach prevents redundant propagation of access paths with identical base variables markedly enhancing analysis performance. Furthermore, we also show this CFL-based approach can enhance the precision beyond that of the traditional access-path-based approach.

To validate our approach, we have integrated it into Heros [Bodden 2012], an IFDS/IDE framework, and developed a new taint analysis tool, named IDEDroid. This tool is designed for detecting information-flow leaks in Android apps, a principal application of the IFDS/IDE framework. Our experimental results demonstrate that IDEDroid not only significantly boosts the efficiency of FlowDroid but also enhances its precision for certain apps. Our main contributions include:

- We present a new approach for alias-aware IFDS analysis, instantiated for taint analysis, by modeling field accesses as edge functions in the IDE framework. This IDE-based analysis significantly enhances both the efficiency and precision of existing IFDS-based analysis.
- We develop a new taint analysis tool (soon-to-be released), namely IDEDroid, built on top of Heros, an IFDS/IDE framework [Bodden 2012].
- We evaluate IDEDroid, equipped with our new CFL-based approach for field-sensitivity, against FlowDroid, which uses Boomerang, a conventional access-path-based approach for field-sensitivity. The evaluation includes 24 major Android apps, with 14 from FossDroid [FossDroid 2023] and 10 from Drebin [Arp et al. 2014; Spreitzenbarth et al. 2013]. IDEDroid achieves an average speedup of 222.0× over FlowDroid, while also reducing average memory usage by 3.6×. Additionally, IDEDroid proves to be more precise than FlowDroid in practice, lowering false positive rates by up to 20.0% in certain apps.

The rest of the paper is organized as follows. Section 2 provides the necessary background on IFDS/IDE. Section 3 motivates our approach with an example. Section 4 formalizes our approach. Section 5 presents and analyzes the experimental results from evaluating IDEDroid against FlowDroid. Section 6 reviews the related work. Finally, Section 7 concludes this paper.

2 Background

The classic IFDS/IDE framework [Sagiv et al. 1996] is a cornerstone in the field of inter-procedural data-flow analysis. IFDS [Reps 1998] frames inter-procedural finite distributive subset data-flow problems as graph-reachability problems. IDE, a generalization of IFDS, is designed to handle inter-procedural distributive environment problems, where data-flow facts are conceptualized as mappings (or environments) from a finite set of symbols to a potentially infinite set of values.

2.1 The IFDS Framework

An IFDS problem IP is defined as a five-tuple $IP = (G^*, D, F, M, \sqcap)$ [Reps 1998]. $G^* = (N^*, E^*)$ represents the supergraph, or the inter-procedural control flow graph (ICFG) of the program. The domain D denotes a finite set of data-flow facts. F, a subset of $2^D \rightarrow 2^D$, comprises a set of flow functions that are distributive over the meet operator $\sqcap$ (either the set union or intersection). Finally, M is a mapping from the edges in E^* to their corresponding flow functions in F.

The supergraph G^* in an IFDS problem comprises a set of control flow graphs (CFGs), $\{G_0, G_1, ...\}$, each representing a different method in the program. In each CFG G_p, there is a unique entry node s_p and a unique exit node e_p. Callsites are depicted with two nodes: a *call* node c_i and a *return* node r_i. There are four types of edges in G^*: *normal edges* representing standard intra-procedural edges; *call edges* connecting c_i to s_p to denote inter-procedural control flow from callsite i to callee p;

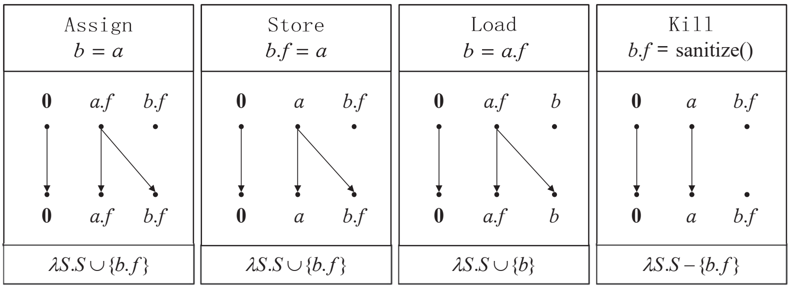


Fig. 1. Flow functions of IFDS-based taint analysis.

return edges linking e_p to r_i for the control flow from callee p back to callsite i; and *call-to-return edges* from c_i to r_i, which convey intra-procedural flow across the callsite that is not part of the call.

The IFDS framework's fundamental concept is transforming an IFDS problem into a graph reachability problem over an exploded graph $G^\# = (N^\#, E^\#)$. This graph, derived from G^*, is defined with $N^\# = N^* \times (D \cup \{\mathbf{0}\})$ and $E^\#$ as $\{\langle m, d_1 \rangle \longrightarrow \langle n, d_2 \rangle \mid m \longrightarrow n \in E^*, d_2 \in M(m,n)(\{d_1\})\}$. Here, $\mathbf{0}$ represents an empty fact, enabling the creation of new data-flow facts. $M(m,n)$ is the flow function for edge (m,n) in $G^\#$, crucial for data-flow fact propagation.

Figure 1 presents the flow functions in IFDS-based taint analysis for four types of statements, focusing on the transformation of tainted access paths (D). It details how each statement's incoming tainted paths are modified by its specific flow function, thus influencing the data-flow facts immediately subsequent to the statement. In a store statement like `b.f = a`, `c.f` becomes tainted as well if `b.f` and `c.f` are aliases. Conversely, `b.f` is untainted following `b.f = sanitize()`, with `sanitize()` indicating a right-hand side that removes taint from `b.f` (if any). This setup illustrates the effect of different statements on the propagation of taint in IFDS-based analysis.

In practical applications, the tabulation algorithm, detailed in [Naeem et al. 2010], is commonly employed for solving IFDS problems. This algorithm operates iteratively, computing path edges until it stabilizes at a fixed point. Within the IFDS terminology, a *path edge* $\langle s_p, d_1 \rangle \longrightarrow \langle n, d_2 \rangle$ indicates a *same-level realizable path* from $\langle s_p, d_1 \rangle$ to $\langle n, d_2 \rangle$, where s_p is the start node of method p containing node n. This means that if data-flow fact d_1 is valid at s_p, then d_2 will be valid at n. Path edges are generated as needed. For example, a path edge $\langle s_p, d_1 \rangle \longrightarrow \langle n, d_2 \rangle$ is established only if $\langle s_p, d_1 \rangle$ can be reached from $s_{\texttt{main}}$ in the main method `main()`. As a result, the path edge $\langle s_p, d_1 \rangle \longrightarrow \langle n, d_2 \rangle$ represents the latter part of a realizable path from $\langle s_{\texttt{main}}, \mathbf{0} \rangle$ to $\langle n, d_2 \rangle$, signifying the propagation of a data-flow fact from the program's entry point to n.

2.2 The IDE Framework

IDE [Sagiv et al. 1996] extends the IFDS framework by allowing the data-flow fact domain to contain environments $env(D, L)$, where D is a finite set of symbols, and L is a finite-height meet lattice. This enhancement enables IDE to not only determine the reachability of a data-flow fact $d \in D$ at a program point, as in IFDS, but also to compute the corresponding value $v \in L$ that d maps to. As a result, IDE is a more general framework than IFDS. In fact, IFDS can be viewed as a specific instance of IDE, where L is a binary domain $\{\top, \bot\}$, representing the binary state of a fact d (e.g., whether it is tainted or not during the IFDS-based analysis) at a program point.

IDE, similar to IFDS, tracks data flow in the exploded supergraph $G^\#$. Each edge in $G^\#$ is associated with an edge function that defines an *environment transformer*. This transformer maps an input environment $env(D, L)$ to an output environment $env(D, L)$, with all transformers being distributive over the meet operator to handle various environment interactions and transformations.

The IDE algorithm operates in two stages, Phase I and Phase II. In Phase I, it calculates same-level realizable paths for each node $\langle n, d\rangle$, which are paths from the entry node of n's method to $\langle n, d\rangle$. This phase also includes summarizing the effects of these paths into a *jump function* by combining the edge functions along the path. Phase II then determines the actual values associated with the nodes in the exploded supergraph. Most of the computational effort in the IDE algorithm is attributed to Phase I, as previously noted in [Naeem et al. 2010] and supported by our experiments. For further details on the IDE algorithm, see [Naeem et al. 2010; Sagiv et al. 1996].

2.3 CFL Reachability

CFL-reachability, as introduced by Reps [1998], establishes a critical framework for formulating a broad spectrum of program analysis problems. This framework identifies a CFL-reachability problem through a context-free language L delineated over an edge-labeled graph G. Two nodes in G are deemed *CFL-reachable* if there exists a path p connecting them, such that the concatenation of the edge labels along p forms a word in L. This mechanism is extensively applied in pointer analysis to ensure field sensitivity [Lu and Xue 2019; Sridharan et al. 2005; Xu et al. 2009; Zheng and Rugina 2008]. By matching field load and store operations, CFL-reachability paths can precisely delineate data flows across mutable heap structures. Nonetheless, existing CFL-based pointer analyses primarily lack flow sensitivity, which compromises their ability to effectively model field kill operations at store statements, essential for representing strong updates on heap data.

3 Motivation

We use taint analysis to motivate our research. Section 3.1 gives a simple program. In Section 3.2, we examine FlowDroid [Arzt et al. 2014], highlighting its access-path-based approach to alias discovery in its IFDS-based taint analysis. In Section 3.3, we introduce our CFL-based approach, offering a more efficient and accurate alternative for alias detection in an IDE-based taint analysis.

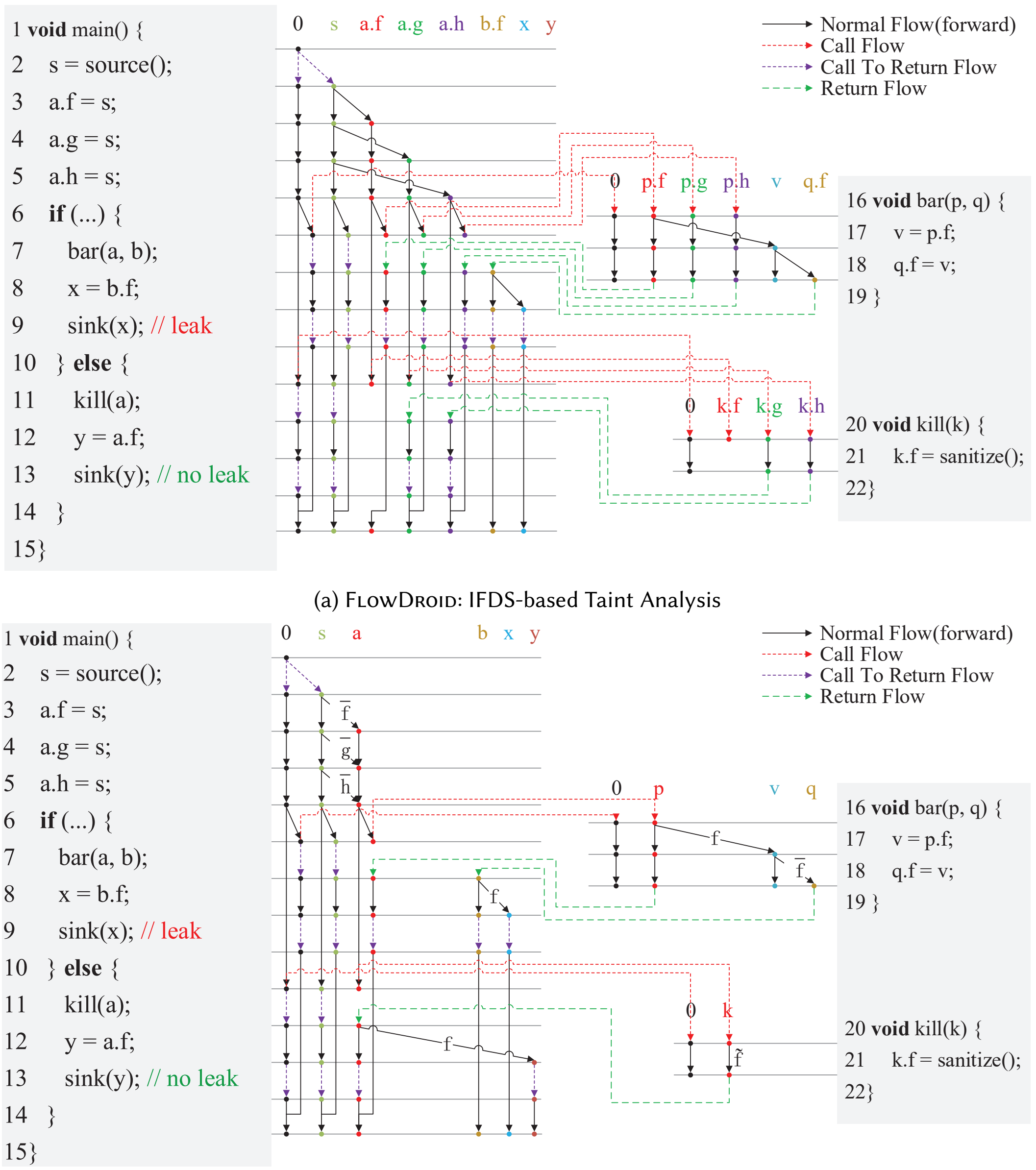
3.1 A Simple Example

In Figure 2, the `main()` method taints variable `s` through `source()` (line 2), which is then stored into `a.f`, `a.g`, and `a.h` (lines 3-5). In the `then` branch (lines 7-9), `a` and `b` are passed to `bar()`'s parameters `p` and `q`. Inside `bar()`, `p.f` is assigned to `q.f` indirectly (lines 17-18), causing `b.f = a.f` and leading to a leak at line 9. Conversely, in the `else` branch (lines 11-13), calling `kill()` (line 11) sanitizes `a.f` (line 21), preventing a leak at line 13. This example is streamlined by excluding inessential elements, such as declaration and object allocation statements.

3.2 FlowDroid: IFDS-based Taint Analysis with Access Paths for Field-Sensitivity

In FlowDroid, the data-flow fact domain D encompasses tainted access paths $v.f^*$. Figure 2(a) gives the exploded supergraph $G^\# = (N^\#, E^\#)$, where a node represents an access path $d \in D$ at a program point, and an edge $\langle n_1, d_1\rangle \longrightarrow \langle n_2, d_2\rangle$ indicates that d_2 at n_2 (i.e., the program point just before line n_2) is tainted by d_1 at n_1 (i.e., the program point just before line n_1). For instance, `a.f = s` at line 3 generates the edge $\langle 3, \mathtt{s}\rangle \longrightarrow \langle 4, \mathtt{a.f}\rangle$. $G^\#$ is built on the fly, with nodes and edges introduced as needed. FlowDroid utilizes a forward solver, propagating tainted access paths until a fixed point. Leak detection hinges on tracing a realizable path to the point of interest. For example, a realizable path from $\langle \mathtt{s}_{\mathtt{main}}, \mathbf{0}\rangle$ to $\langle 9, \mathtt{x}\rangle$, $\langle \mathtt{s}_{\mathtt{main}}, \mathbf{0}\rangle \longrightarrow \langle 2, \mathbf{0}\rangle \longrightarrow \langle 3, \mathtt{s}\rangle \longrightarrow \langle 4, \mathtt{a.f}\rangle \longrightarrow \langle 7, \mathtt{a.f}\rangle \longrightarrow \langle 17, \mathtt{p.f}\rangle \longrightarrow \langle 18, \mathtt{v}\rangle \longrightarrow \langle 19, \mathtt{q.f}\rangle \longrightarrow \langle 8, \mathtt{b.f}\rangle \longrightarrow \langle 9, \mathtt{x}\rangle$ signals a leak at line 9.

FlowDroid offers two demand-driven approaches for identifying tainted field value aliases at store statements, such as `q.f = v` at line 18. Its default option leverages a backward solver to navigate reverse control flow, detecting aliases (e.g., `b.f` as an alias of `q.f`) and injecting them into FlowDroid's forward solver to aid in leak detection, as seen at line 9. Alternatively, Boomerang [Späth



```
void main() {
  s = source();
  a.f = s;
  a.g = s;
  a.h = s;
  if (...) {
    bar(a, b);
    x = b.f;
    sink(x); // leak
  } else {
    kill(a);
    y = a.f;
    sink(y); // no leak
  }
}
```

```
void bar(p, q) {
  v = p.f;
  q.f = v;
}
```

```
void kill(k) {
  k.f = sanitize();
}
```

(a) FlowDroid: IFDS-based Taint Analysis

```
void main() {
  s = source();
  a.f = s;
  a.g = s;
  a.h = s;
  if (...) {
    bar(a, b);
    x = b.f;
    sink(x); // leak
  } else {
    kill(a);
    y = a.f;
    sink(y); // no leak
  }
}
```

```
void bar(p, q) {
  v = p.f;
  q.f = v;
}
```

```
void kill(k) {
  k.f = sanitize();
}
```

(b) IDEDroid: IDE-based Taint Analysis

Fig. 2. Comparing IDEDroid and FlowDroid for performing taint analysis.

et al. 2016] adopts a bi-directional IFDS analysis at store statements to uncover aliased paths like {b.f}, which are then processed by FlowDroid's forward solver for forward propagation. Both strategies offer equivalent precision in analysis with minor differences in performance. The alias identification mechanism, not central to this work, is omitted from Figure 2 for clarity.

We have identified three critical limitations in FlowDroid that significantly affect its performance, especially when utilizing the access-path-based approach for achieving field-sensitivity:

- **Large Data-Flow Fact Domain.** In FlowDroid, access paths, such as `a.f`, `a.g`, and `a.h` in Figure 2(a), are encoded as data-flow facts in the exploded graph $G^{\#}$, at every program point. Maintaining access paths explicitly for field-sensitivity causes a combinatorial growth in data-flow facts, substantially expanding D. For some programs, the number of maintained access paths can be an order of magnitude larger than the base variables (Figure 10).
- **Redundant Taint Propagation.** In FlowDroid, many tainted access paths are propagated unnecessarily, notably affecting performance. For example, during the call to `bar(a,b)` (line 7), the three tainted paths `a.f`, `a.g`, and `a.h` are propagated to `bar()`, resulting in `p.f`, `p.g`, and `p.h` being further propagated within this callee. However, propagating `a.g` and `a.h` into `bar()` is redundant as they are not accessed in the callee. This unavoidable over-propagation, due to a lack of early insight, leads to performance degradation.
- **Redundant Alias Query.** In FlowDroid, an alias query is initiated when a field store is tainted. For instance, in our example, if line 18 is modified to `q.f = p`, and since `p.f`, `p.g`, and `p.h` are tainted, then three alias queries for `q.f.f`, `q.f.g`, and `q.f.h` with the same base variable `q` are triggered at this store statement. While Boomerang attempts to minimize redundant alias queries through caching, this approach demands additional data structures. Moreover, cached results are susceptible to being flushed by other unrelated queries, potentially limiting the effectiveness of this caching strategy.

3.3 IDEDroid: IDE-Based Taint Analysis with CFL-Based Environment Transformers

To overcome the three limitations in FlowDroid's field-sensitive approach, we bypass generating explicit access paths, reframing it as a CFL-reachability problem in the IDE framework. Our approach symbolically encodes access paths as edge functions or environment transformers. Rather than computing tainted access paths at each edge, its transformer maps a base variable $v \in D$ to the set of its field access sequences in $l \in L$, where $v.f^*$ (with $f^* \in l$) is an access path on the base variable v. Here, L is a finite-height lattice of field access sequences, using set union as the meet operator.

Figure 2(b) illustrates IDEDroid, a novel IDE-based taint analysis, which adopts a CFL-based strategy for field-sensitivity. Unlike FlowDroid, IDEDroid only considers base variables such as `a`, `p`, `q`, and `k` as data-flow facts (nodes) in the exploded graph $G^{\#}$. These variables' access paths are computed using environment transformers by solving a CFL-reachability problem pertaining to their field access sequences. The basic idea behind our CFL-based approach is explained below.

In our approach, we categorize field accesses f into three types: $\overline{f}$ for field store, f for field load, and $\widetilde{f}$ for field sanitization. For instance, in Figure 2(b), an edge like $\mathtt{s} \xrightarrow{\overline{\mathtt{f}}} \mathtt{a}$ represents a field store (`a.f = s` at line 3), while $\mathtt{b} \xrightarrow{\mathtt{f}} \mathtt{x}$ indicates a field load (`x = b.f` at line 8). Additionally, an edge like $\mathtt{k} \xrightarrow{\widetilde{\mathtt{f}}} \mathtt{k}$ signifies the sanitization or killing of the access path `k.f` (`k.f = sanitize()` at line 21).

The path $\langle \mathtt{s}_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle 2, \mathbf{0}\rangle \rightarrow \langle 3, \mathtt{s}\rangle \xrightarrow{\overline{\mathtt{f}}} \langle 4, \mathtt{a}\rangle \rightarrow \langle 7, \mathtt{a}\rangle \rightarrow \langle 17, \mathtt{p}\rangle \xrightarrow{\mathtt{f}} \langle 18, \mathtt{v}\rangle \xrightarrow{\overline{\mathtt{f}}} \langle 19, \mathtt{q}\rangle \rightarrow \langle 8, \mathtt{b}\rangle \xrightarrow{\mathtt{f}} \langle 9, \mathtt{x}\rangle$ is CFL-reachable in IDEDroid, resulting in a leak detection at line 9 as the sequence of edge labels $\overline{\mathtt{f}}\mathtt{f}\overline{\mathtt{f}}\mathtt{f}$ appears in our CFL. However, the path $\langle \mathtt{s}_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle 2, \mathbf{0}\rangle \rightarrow \langle 4, \mathtt{s}\rangle \xrightarrow{\overline{\mathtt{g}}} \langle 5, \mathtt{a}\rangle \rightarrow \langle 7, \mathtt{a}\rangle \rightarrow \langle 17, \mathtt{p}\rangle \xrightarrow{\mathtt{f}} \langle 18, \mathtt{v}\rangle \xrightarrow{\overline{\mathtt{f}}} \langle 19, \mathtt{q}\rangle \rightarrow \langle 8, \mathtt{b}\rangle \xrightarrow{\mathtt{f}} \langle 9, \mathtt{x}\rangle$ is not CFL-reachable, as the sequence $\overline{\mathtt{g}}\mathtt{f}\overline{\mathtt{f}}\mathtt{f}$ does not conform to our CFL requirements. In addition, the path $\langle \mathtt{s}_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle 2, \mathbf{0}\rangle \rightarrow \langle 3, \mathtt{s}\rangle \xrightarrow{\overline{\mathtt{f}}} \langle 4, \mathtt{a}\rangle \rightarrow \langle 11, \mathtt{a}\rangle \rightarrow \langle 21, \mathtt{k}\rangle \xrightarrow{\widetilde{\mathtt{f}}} \langle 22, \mathtt{k}\rangle$ is discarded by $\widetilde{\mathtt{f}}$ (as the nearest unmatched store is $\overline{\mathtt{f}}$), effectively preventing tainting of `b.f`, and consequently, ruling out a leak at line 13.

IDEDroid, operating in the IDE framework, limits propagation to base variables in the exploded graph during Phase I. This phase involves calculating jump functions (intra-procedural environment transformers) at every program point. For example, at the exit node of `bar()`, the jump function is $\mathtt{p} \xrightarrow{\mathtt{f}\bar{\mathtt{f}}} \mathtt{q}$. This strategy effectively reduces superfluous taint propagation and alias queries. Overall, our IDE-based technique for field-sensitivity offers a number of key advantages:

- **Smaller Data-flow Fact Domain.** Utilizing a CFL to define field accesses allows D to comprise only base variables, significantly smaller than the version of D in FlowDroid's access-path-based approach, which includes all access paths in the program (Figure 10).
- **Less Taint Propagation.** IDEDroid's taint analysis in the IDE framework comprises two phases: Phase I, where it iteratively computes *jump functions* (intra-procedural environment transformers) until a fixed point, and Phase II, which calculates actual values, i.e., access paths for each base variable v. The time-intensive Phase I is optimized by exclusively handling base variables, thus avoiding unnecessary propagation of access paths.
- **Fewer Alias Queries.** In contrast to FlowDroid, which issues a separate alias query for each unique taint source at the same field store, IDEDroid's CFL-based approach limits alias queries to Phase I, generating just one query per field store. This strategy effectively reduces the necessity for multiple queries for the same field store affected by different tainted access paths, a common issue in FlowDroid discussed in Section 3.2.

```
1 void main() {                 7 foo(p) {
2   b1.h = source();            8   x.f1.f2 = p;
3   b2 = foo(b1);               9   y = x.f1.f2;
4   c = b2.g;                  10   return y;
5   sink(c);//no leak          11 }
6 }
```

Fig. 3. Precision improvement of IDEDroid over FlowDroid (with 2-limiting).

- **Improved Precision.** Our CFL-based approach always outperforms the access-path-based alternative in precision under k-limiting with the same k value. For example, consider a scenario with 2-limiting illustrated in Figure 3. FlowDroid, unable to differentiate between `x.f1.f2.h` and `x.f1.f2.g` (both abstracted as `x.f1.f2`), erroneously reports a leak at line 5. In contrast, our approach simplifies $\langle 8, \mathtt{p}\rangle \xrightarrow{\overline{\mathtt{f2}}\,\overline{\mathtt{f1}}} \langle 9, \mathtt{x}\rangle \xrightarrow{\mathtt{f1}\ \mathtt{f2}} \langle 10, \mathtt{y}\rangle$ in `foo()` to $\langle 8, \mathtt{p}\rangle \longrightarrow \langle 10, \mathtt{y}\rangle$ as the function summary, balancing all edge labels. When analyzing `main()`, a summary edge $\langle 3, \mathtt{b1}\rangle \longrightarrow \langle 4, \mathtt{b2}\rangle$ is created due to the call to `foo()` at Line 3. Since the path $\langle 2, \mathbf{0}\rangle \xrightarrow{\bar{\mathtt{h}}} \langle 3, \mathtt{b1}\rangle \longrightarrow \langle 4, \mathtt{b2}\rangle \xrightarrow{\mathtt{g}} \langle 5, \mathtt{c}\rangle$ is not CFL-reachable (as $\bar{\mathtt{h}}\mathtt{g}$ does not balance out), IDEDroid correctly identifies that there is no leak at line 5.

4 Methodology

We describe our CFL-based field-sensitivity approach in the context of our new taint analysis tool, IDEDroid, developed within the IDE framework. The tool's workflow is depicted in Figure 4. In this setup, the IDE solver computes environments mapping base variables (D) to sets of field access strings (L), essentially denoting field access sequences. IDEDroid processes base variables as data-flow facts in the program's exploded graph $G^{\#}$, while tracking field accesses along each program path. These field accesses are then evaluated through CFL-reachability, as shown in Figures 6 and 7.

IDEDroid uses Boomerang [Späth et al. 2016], a demand-driven, flow-, field-, and context-sensitive pointer analysis, for alias queries at field stores of base variables in D. It is important to note that the alias analysis is orthogonal to this work. Boomerang returns aliased access paths,

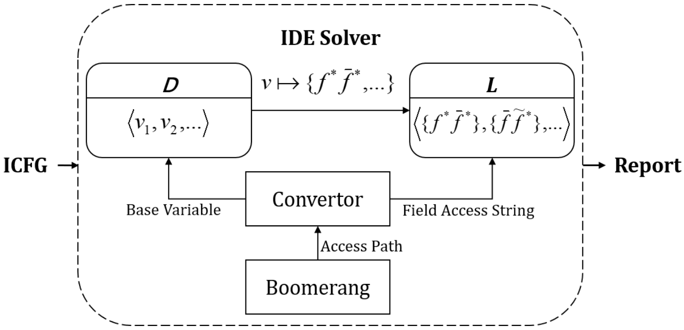

IDE Solver
D
$\langle v_1, v_2, \ldots \rangle$
$v \mapsto \{f^* \bar{f}^*, \ldots\}$
L
$\langle \{f^* \bar{f}^*\}, \{\bar{f} \tilde{f}^*\}, \ldots \rangle$
ICFG
Report
Base Variable
Convertor
Field Access String
Access Path
Boomerang

Fig. 4. The IDEDroid workflow in the IDE framework.

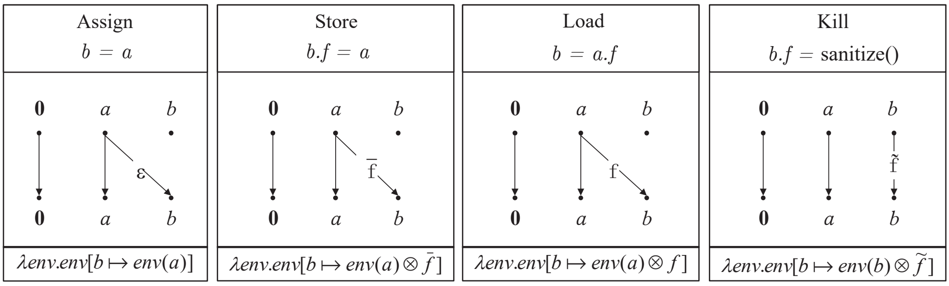

Assign
$b = a$
Store
$b.f = a$
Load
$b = a.f$
Kill
$b.f =$ sanitize()
0
a
b
ε
$\bar{f}$
f
$\tilde{f}$
$\lambda env.env[b \mapsto env(a)]$
$\lambda env.env[b \mapsto env(a) \otimes \bar{f}]$
$\lambda env.env[b \mapsto env(a) \otimes f]$
$\lambda env.env[b \mapsto env(b) \otimes \tilde{f}]$

Fig. 5. Environment transformers used in IDEDroid.

which are transformed by IDEDroid into pairs $\langle v, f^* \rangle$ (with v as a base variable and f^* as a field access sequence). For example, `v.f.g` becomes $\langle \mathtt{v}, \bar{\mathtt{g}}\bar{\mathtt{f}} \rangle$, with $\bar{\mathtt{f}}$ and $\bar{\mathtt{g}}$ indicating field stores. This updates D and L accordingly. Following [Sagiv et al. 1996], IDEDroid operates in two phases: Phase I propagates base variables and computes jump functions in the program's exploded graph using alias information, and Phase II identifies actual field access sequences for each base variable.

4.1 Environments and Environment Transformers for Taint Analysis

As outlined in [Sagiv et al. 1996], the set of environments $Env(D, L)$ consists of functions mapping from D to L, where D is a finite set of symbols, and L is a finite-height meet semi-lattice. An environment $env(D, L) \in Env(D, L)$ maps a symbol $d \in D$ to a value $l \in L$, represented as $env(d) \mapsto l$. The meet operator for $Env(D, L)$, expressed as $env_1 \sqcap env_2$, is defined as $\lambda d.(env_1(d) \sqcap env_2(d))$.

In the case of taint analysis, D represents the set of base variables, and L is the semi-lattice consisting of sets of *field access strings* with set union as the meet operator $\sqcap$. A field access string is a labeled string recording field accesses along a program path. An environment $env(D, L)$ maps a base variable $v \in D$ to a set of field access strings $l \in L$, collectively representing access paths with the base variable v. Rather than encoding an access path like $v.f$ directly as $env(v) \mapsto \{f\}$, l includes different types of field accesses—stores, loads, and kills (sanitization) of field f—essential for computing jump functions in Phase I. As briefly observed from Figure 5, which will be elaborated on shortly, labels $\bar{f}$, f, and $\tilde{f}$ are used for store, load, and kill, respectively.

The environment $env(D, L)$ is formally defined as follows.

Definition 4.1. *In IDEDroid, $env(D, L)$ associates each base variable D with a set L of field access strings, where each string $l \in L$ consists of labels $\bar{f}$, f, $\tilde{f}$, or ϵ for the empty string.*

$$
\begin{aligned}
\mathcal{F} &::= \mathcal{L}\,\mathcal{S} \\
\mathcal{L} &::= \mathcal{M}\,\mathcal{L} \mid f_i\,\mathcal{L} \mid \epsilon \\
\mathcal{S} &::= \mathcal{M}\,\mathcal{S} \mid \overline{f_i}\,\mathcal{S} \mid \epsilon \\
\mathcal{M} &::= \overline{f_i}\,\mathcal{M}\,f_i \mid \mathcal{M}\,\mathcal{M} \mid \epsilon
\end{aligned}
$$

Fig. 6. A simplified context-free grammar for field accesses without considering kill operations.

We introduce ⊗, a left-associative binary operator, efficiently implemented as described in Section 4.2. It operates on both individual and sets of field access strings, concatenating them and then verifying their validity against our CFL. Based on CFL-reachability, outcomes are either updated or discarded. When applied to sets, ⊗ evaluates each pair in their Cartesian product.

An environment transformer $t : Env(D, L) \mapsto Env(D, L)$ is a function that maps one environment to another. Within the IDE framework, these transformers are linked to edges in the program's exploded graph $G^{\#}$, by assigning an *edge function*, $EdgeFn^{d}_{m,n}$, to each edge $\langle m, d\rangle \rightarrow \langle n, d'\rangle$. This edge function maps L to L, so that $d \xrightarrow{EdgeFn^{d}_{m,n}} d'$ denotes the transformer $\lambda env.env[d' \mapsto EdgeFn^{d}_{m,n}(env(d))]$, updating the target fact d' based on the source fact d.

Figure 5 gives four types of edge functions and their environment transformers employed by IDEDroid for four different categories of statements. As these edge functions are simple, when writing $\langle m, a\rangle \xrightarrow{l} \langle n, b\rangle$ (with l as a set of field access strings), we imply an edge function $EdgeFn(env(a)) = env(a) \otimes l$, streamlining the transformer to $\lambda env.env[b \mapsto env(a) \otimes l]$. For the singleton set $l = \{f^*\}$, we sometimes write f^* directly. Let us take a closer look at Figure 5 below:

- **Assignment:** An edge $\langle m, a\rangle \rightarrow \langle n, b\rangle$ for assignment $b = a$ uses the transformer $\lambda env.env[b \mapsto env(a)]$. This copies field access strings from a to b.
- **Store:** A store edge $\langle m, a\rangle \xrightarrow{\overline{f}} \langle n, b\rangle$ for $b.f = a$ has the transformer $\lambda env.env[b \mapsto env(a) \otimes \overline{f})]$. Additionally, for an aliased access path of $b.f$ like $c.g.h$, an edge $\langle m, a\rangle \xrightarrow{\overline{h}\overline{g}} \langle n, c\rangle$ is added, denoted by $\lambda env.env[c \mapsto env(a) \otimes \overline{h}\overline{g}]$.
- **Load and Kill:** For a load $\langle m, a\rangle \xrightarrow{f} \langle n, b\rangle$ ($b = a.f$) and a kill $\langle m, b\rangle \xrightarrow{\widetilde{f}} \langle n, b\rangle$ ($b.f = sanitize()$), the transformers set b's field access string set to $env(a) \otimes f$ and $env(b) \otimes \widetilde{f}$, respectively, while keeping everything else unchanged.

The transformers described above modify $env(b)$ using the values from $env(a)$. It is straightforward to confirm that all the environment transformers shown in Figure 5 are distributive.

4.2 CFL-Reachability for Field-Sensitivity

A new CFL is defined with a context-free grammar specifically designed for capturing *valid* field access strings, labeling any string not conforming to this grammar as *invalid*, subject to removal. We start with a simplified version in Figure 6, where kill operations are overlooked by treating $\widetilde{f}$ as ϵ. In this grammar, the terminals (or labels) $\overline{f_i}$, f_i, and ϵ symbolize store, load, and assign operations, respectively. In addition, the non-terminal symbols are defined as follows:

- $\mathcal{M}$: This symbolizes a string with perfectly matched store ($\overline{f_i}$) and load (f_i) accesses.
- $\mathcal{L}$: Strings derived as $\mathcal{L}$ have matched stores (if present) and may end with field loads.
- $\mathcal{S}$: Strings derived as $\mathcal{S}$ have matched loads (if present) and may end with field stores.
- $\mathcal{F}$: The start symbol, an $\mathcal{F}$-string, results from concatenating an $\mathcal{L}$-string with an $\mathcal{S}$-string.

By convention, a string derived from a nonterminal X is referred to as an *X-string*.

$$
\begin{aligned}
&\mathcal{F} ::= \mathcal{L}\,\mathcal{K}\,\mathcal{S}\\
&\mathcal{L} ::= \mathcal{M}\,\mathcal{L} \mid \mathcal{L}_i\,\mathcal{L} \mid \epsilon\\
&\mathcal{S} ::= \mathcal{M}\,\mathcal{S} \mid \mathcal{S}_i\,\mathcal{S} \mid \epsilon\\
&\mathcal{M} ::= \mathcal{S}_i\,\mathcal{L}_i \mid \mathcal{M}\,\mathcal{M} \mid \epsilon\\
&\mathcal{K} ::= \widetilde{f_i}\,\mathcal{K} \mid \epsilon\\
&\mathcal{L}_i ::= f_i\,\mathcal{M} \mid \widetilde{f_j}\,\mathcal{L}_{i(i\neq j)}\\
&\mathcal{S}_i ::= \overline{f_i}\,\mathcal{M} \mid \mathcal{S}_i\,\widetilde{f}_{j(i\neq j)}
\end{aligned}
$$

Fig. 7. A complete context-free grammar for specifying field accesses in IDEDroid.

Without kill operations, a taint propagation path through a series of statements is captured by a field access string comprising a sequence of load labels followed by store labels. Any perfectly matched pairs of stores and loads simplify to ϵ, reflecting removal of balanced paratheses.

The grammar in Figure 6 defines an extended Dyck CFL for partially matched fields [Kodumal and Aiken 2004; Shi et al. 2022]. In our motivating example from Figure 2, $\mathtt{f}\overline{\mathtt{f}}$ is a valid partially matched string on the path $\langle 17, \mathtt{p}\rangle \xrightarrow{\mathtt{f}} \langle 18, \mathtt{v}\rangle \xrightarrow{\overline{\mathtt{f}}} \langle 19, \mathtt{q}\rangle$, indicating that `v` is first loaded from `p.f` and then stored to `q.f`. Conversely, a string like $\overline{\mathtt{g}}\mathtt{f}$ is invalid according to the grammar, as it implies loading from field `f`, which does not match the previously stored field $\overline{\mathtt{g}}$.

4.2.1 Modeling Kill Operations. The grammar in Figure 6 is expanded in Figure 7 to include kill operations. $\mathcal{L}_i$ denotes field access strings with a single unmatched load lacking an immediate left-matching kill, because a kill $\widetilde{f}$ nullifies a string if followed by a corresponding load f (e.g., `b.f = sanitize()` sanitizes a subsequent load `x = b.f`). Similarly, $\mathcal{S}_i$ represents field access strings with a single unmatched store without an immediate right-matching kill, as a kill $\widetilde{f}$ voids a string if it follows a matching store $\overline{f}$ (e.g., `b.f = sanitize()` sanitizes a preceding store `b.f = s`).

In the revised grammar of Figure 7, $\mathcal{L}$, $\mathcal{S}$, and $\mathcal{M}$'s productions were updated, replacing $\overline{f_i}$ and f_i with $\mathcal{S}_i$ and $\mathcal{L}_i$. This adjustment allows $\mathcal{L}$ and $\mathcal{S}$ to represent sequences of loads and stores, including kills, once matched pairs are simplified to ϵ. Thus, $\mathcal{M}$ accurately reflects strings of matched stores and loads, incorporating kills. In taint propagation, however, kills from $\mathcal{L}$, $\mathcal{S}$, or $\mathcal{M}$ are deemed inconsequential, treated as ϵ, since they do not affect taint propagation. For example, the code "`a = source(); a.f = sanitize(); b = a.g; c = b.f;`" produces a string $\widetilde{\mathtt{f}}\mathtt{gf}$ from $\mathcal{L}$, where $\widetilde{\mathtt{f}}$ is seen as ϵ. Likewise, "`a = source(); b.g = a; b.f = sanitize(); c.f = b;`" generates a string $\overline{\mathtt{g}}\widetilde{\mathtt{f}}\overline{\mathtt{f}}$ from $\mathcal{S}$, with $\widetilde{\mathtt{f}}$ also regarded as ϵ.

Therefore, to account for kill operations in taint propagation, a new non-terminal symbol $\mathcal{K}$ is introduced, signifying strings of any number of kills. An $\mathcal{F}$-string combines an $\mathcal{L}$-string, a $\mathcal{K}$-string, and an $\mathcal{S}$-string, placing $\mathcal{K}$ between $\mathcal{L}$ and $\mathcal{S}$ to effectively capture kills' impact.

With this modeling of kill operations, a taint propagation path is represented by a field access string consisting of sequences of loads ($\mathcal{L}$), kills ($\mathcal{K}$), and stores ($\mathcal{S}$). It is understood that kills within $\mathcal{L}$ and $\mathcal{S}$ are disregarded, and balanced pairs of stores and loads are reduced to ϵ.

In the code sequence "1: `b.f1 = a`; 2: `b.f1 = sanitize()`; 3: `c = b.f1`; 4: …", where `a` is assumed to be already tainted, the path $\langle 1, \mathtt{a}\rangle \xrightarrow{\overline{\mathtt{f1}}} \langle 2, \mathtt{b}\rangle \xrightarrow{\widetilde{\mathtt{f1}}} \langle 3, \mathtt{b}\rangle \xrightarrow{\mathtt{f1}} \langle 4, \mathtt{c}\rangle$ forms $\overline{\mathtt{f1}}\widetilde{\mathtt{f1}}\mathtt{f1}$, invalidated by the grammar in Figure 7. This indicates "`b.f1 = sanitize()`" sanitizes the tainted path through `b.f1`. Modifying line 2 to "`b.f2 = sanitize()`" results in $\overline{\mathtt{f1}}\widetilde{\mathtt{f2}}\mathtt{f1}$, which the grammar accepts (as a balanced parenthesis according to the grammar in Figure 7), showing `c` becomes tainted from `b.f1`, unaffected by the unrelated `b.f2` sanitization (with $\widetilde{\mathtt{f2}}$ interpreted as ϵ). Finally, consider "`a = source(); b = a.f; c = b.g; c.h = sanitize(); x.r = c; y = x.r; z = y.h;`" Given its field access string as $\mathtt{fg}\widetilde{\mathtt{h}}\overline{\mathtt{r}}\mathtt{rh}$, `z` is untainted as `c.h`, i.e., `y.h` has been effectively sanitized.

4.2.2 Efficient CFL-Reachability Solving. We have devised an efficient algorithm to analyze field accesses based on CFL-reachability, building upon the following four key observations:

- The $\otimes$ operator is applied only to an $\mathcal{F}$-string F and a terminal symbol. For the operation $F \otimes F'$, with $F' = f_1 ... f_k$, this is equivalent to sequential applications $F \otimes f_1 \otimes \ldots \otimes f_k$.
- For an $\mathcal{F}$-string F, applying $F \otimes \overline{f}$ invariably results in an $\mathcal{F}$-string. $F \otimes \widetilde{f}$ produces an $\mathcal{F}$-string if $\widetilde{f}$ does not match the last store in F. Conversely, $F \otimes f$ yields an $\mathcal{F}$-string if f matches the last store $\overline{f}$ in F or if F has no stores and f does not match any kill in F.
- In an $\mathcal{F}$-string F, the order in which the kill operations appear does not influence the outcomes of subsequent operations. Whether $F \otimes \overline{f}$ or $F \otimes \widetilde{f}$, the kill operation $\widetilde{f}$, once added to F, will not sanitize any existing field accesses in F. For instance, the sequence "`b = source(); a.f1 = b; x.f2 = a; x.f1 = sanitize()`" illustrates that the kill $\widetilde{\texttt{f1}}$ fails to cleanse the taint on `a.f1`, since `x.f1` and `a.f1` have distinct base variables. However, for $F \otimes f$, it is crucial to examine all kill operations in F regarding f to evaluate their effectiveness in field load sanitation. The scenario "`x.f1 = sanitize(); x.f2 = sanitize(); y = x.f1`" exemplifies a successful kill operation, effectively rendering `y` untainted.
- In an $\mathcal{F}$-string, a pair of matching load and store operations can be simplified to ϵ.

For an $\mathcal{F}$-string F, since the order of kill operations does not matter, we can represent F as a sequence of loads and stores, alongside a distinct set of kills. By reducing matched load and store pairs to ϵ, we can simplify this particular sequence into a sub-sequence of unmatched loads followed by a sub-sequence of unmatched stores. This streamlined approach allows us to improve the efficiency of $\otimes$ operations by offering a more concise representation of F.

Definition 4.2. *A valid field access string F defined by the grammar in Figure 7 is expressed by a triple* $\langle L_F, K_F, S_F \rangle$, *where* $L_F = f_i^*$, $K_F = \{\widetilde{f_0}, \widetilde{f_1}, ...\}$, *and* $S_F = \overline{f_i}^*$.

From the above observations, we can specifically define $\otimes$ for pairs of a field access string and a terminal, enabling these operations to be executed efficiently in nearly constant time:

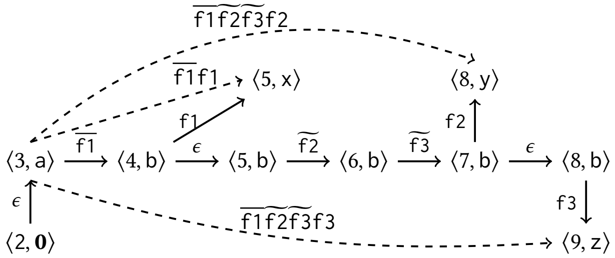
$$\bigotimes: \quad \begin{cases} F \bigotimes f \quad = \begin{cases} \textsc{Nil} & S_F = \epsilon \wedge \widetilde{f} \in K_F \\ \langle cons(f, L_F), \emptyset, S_F \rangle & S_F = \epsilon \wedge \widetilde{f} \notin K_F \\ \textsc{Nil} & car(S_F) = \overline{f'} \quad (f \neq f') \\ \langle L_F, K_F, cdr(S_F) \rangle & car(S_F) = \overline{f} \end{cases} \\ \\ F \bigotimes \widetilde{f} \quad = \begin{cases} \textsc{Nil} & car(S_F) = \overline{f} \\ \langle L_F, K_F, S_F \rangle & car(S_F) = \overline{f'} \quad (f \neq f') \\ \langle L_F, K_F \cup \{\widetilde{f}\}, S_F) \rangle & S_F = \epsilon \end{cases} \\ \\ F \bigotimes \overline{f} \quad = \quad \langle L_F, K_F, cons(\overline{f}, S_F) \rangle \end{cases} \tag{1}$$

This definition employs the standard *cons*, *car*, and *cdr* functions, outlining three $\otimes$ operations: $F \otimes f$, $F \otimes \widetilde{f}$, and $F \otimes \overline{f}$. The function $\text{cons}(f, F)$ constructs a new access string by prepending f to F. The function $\text{car}(F)$ returns the first element of F, and the function $\text{cdr}(F)$ returns the remainder of F, excluding the first element. For each operation, where F represents a valid F-string, the outcome must also be a valid string, unless it is discarded during analysis. Let us examine these operations further.

The $F \otimes f$ operation is the most complex, unfolding in four cases. Initially, if F lacks stores ($S_F = \epsilon$) and f is already countered by a prior kill ($\widetilde{f} \in K_F$), the resulting string is disregarded (signified by Nil). Next, when F is devoid of stores and f escapes neutralization by any kill in F ($f \notin K_F$), f

```
1 void main() {
2   a = source();
3   b.f1 = a;
4   x = b.f1;
5   b.f2 = sanitize() ;
6   b.f3 = sanitize() ;
7   y = b.f2;
8   z = b.f3;
9 }
```

(a) An example program


$\overline{\mathtt{f1}}\widetilde{\mathtt{f2}}\widetilde{\mathtt{f3}}\mathtt{f2}$
$\overline{\mathtt{f1}}\mathtt{f1}$
$\langle 5, \mathtt{x}\rangle$
$\langle 8, \mathtt{y}\rangle$
$\mathtt{f1}$
$\mathtt{f2}$
$\langle 3, \mathtt{a}\rangle$
$\overline{\mathtt{f1}}$
$\langle 4, \mathtt{b}\rangle$
ϵ
$\langle 5, \mathtt{b}\rangle$
$\widetilde{\mathtt{f2}}$
$\langle 6, \mathtt{b}\rangle$
$\widetilde{\mathtt{f3}}$
$\langle 7, \mathtt{b}\rangle$
ϵ
$\langle 8, \mathtt{b}\rangle$
ϵ
$\mathtt{f3}$
$\overline{\mathtt{f1}}\widetilde{\mathtt{f2}}\widetilde{\mathtt{f3}}\mathtt{f3}$
$\langle 2, \mathbf{0}\rangle$
$\langle 9, \mathtt{z}\rangle$

(b) Part of its exploded supergraph

Fig. 8. An example for illustrating our CFL-based approach utilizing the ⊗ operations in Equation (1).

gets added to F, and K_F is cleared to $\emptyset$, indicating a kill only affects its immediate right-matching load. If f mismatches the nearest store in F ($car(S_F) = \overline{f'}$ such that $f' \neq f$), this configuration is also discarded, signifying an unreachable CFL path. Conversely, if f matches the nearest store ($car(S_F) = \overline{f}$), that matching store is removed from F, simplifying the load and store pair to ϵ.

For the operation $F\otimes\widetilde{f}$, if $\widetilde{f}$ matches the nearest store $car(S_F)$, the string is discarded. This signifies that a kill operation only impacts the store immediately preceding it that matches, indicating the sanitization of the field f. If there is no matching store, F remains the same. When F lacks stores, $\widetilde{f}$ is appended to F, setting the stage to sanitize any future load (captured by the first case in $F\otimes f$).

For the operation $F\otimes\overline{f}$, which is applied for processing a store statement, $\overline{f}$ is simply added to F.

For these three ⊗ operations, all except $F\otimes f$ execute in $O(1)$ time. The worst-case time complexity for $F\otimes f$, in cases where $\widetilde{f} \in K_F$ and $\widetilde{f} \notin K_F$, scales linearly with the number of kill operations in K_F. However, on average, these cases can be processed in $O(1)$ due to K_F being a hash set.

Revisiting the four edge functions depicted in Figure 5, for edges $\langle m, a\rangle \xrightarrow{l} \langle n, b\rangle$ in $G^{\#}$, the environment transformers take the form $\lambda env.env[b \mapsto env(a)\otimes l]$, where l is a set of field access strings. In the case of assignments, where $l = \epsilon$, the transformer simplifies to $\lambda env.env[b \mapsto env(a)\otimes\epsilon]$, i.e., $\lambda env.env[b \mapsto env(a)]$. For store operations indicated by $\langle m, a\rangle \xrightarrow{\overline{f}} \langle n, b\rangle$, transformers are defined as $\lambda env.env[b \mapsto env(a)\otimes\overline{f}]$, where $\{\overline{f}\}$ is notationally reduced to $\overline{f}$ for clarify.

In IDEDroid, taint sources are processed in a standard fashion (at least conceptually). For example, given the statement “ℓ: `x = source()`”, located at line number ℓ, an edge $\langle \ell, \mathbf{0}\rangle \xrightarrow{\epsilon} \langle \ell + 1, x\rangle$ is added to the exploded supergraph of the program. This indicates that the variable `x` is tainted just prior to line $\ell + 1$. According to Definition 4.2, this ϵ is represented by the triple $\langle \epsilon, \emptyset, \epsilon\rangle$.

Figure 8 illustrates an application of Equation (1) to a simple example. By analyzing $\langle 2, \mathbf{0}\rangle \xrightarrow{\epsilon} \langle 3, \mathtt{a}\rangle \xrightarrow{\overline{\mathtt{f1}}} \langle 4, \mathtt{b}\rangle \xrightarrow{\mathtt{f1}} \langle 5, \mathtt{x}\rangle$, we identify the following: the $\mathcal{F}$-string $\langle \epsilon, \emptyset, \epsilon\rangle$ for `a` before line 3; $\langle \epsilon, \emptyset, \overline{\mathtt{f1}}\rangle$ for `b` before line 4; and $\langle \epsilon, \emptyset, \epsilon\rangle$ for `x` before line 5. This analysis indicates that `x = b.f1` is tainted just before line 5, as further discussed in Section 4.3. In the extended path leading to `y`, $\langle 2, \mathbf{0}\rangle \xrightarrow{\epsilon} \langle 3, \mathtt{a}\rangle \xrightarrow{\overline{\mathtt{f1}}} \langle 4, \mathtt{b}\rangle \xrightarrow{\epsilon} \langle 5, \mathtt{b}\rangle \xrightarrow{\widetilde{\mathtt{f2}}} \langle 6, \mathtt{b}\rangle \xrightarrow{\widetilde{\mathtt{f3}}} \langle 7, \mathtt{b}\rangle \xrightarrow{\mathtt{f2}} \langle 8, \mathtt{y}\rangle$, we deduce an $\mathcal{F}$-string $\langle \epsilon, \{\widetilde{\mathtt{f2}}, \widetilde{\mathtt{f3}}\}, \epsilon\rangle$ before line 7. This string transforms to Nil for `y` before line 8 after computing $\langle \epsilon, \{\widetilde{\mathtt{f2}}, \widetilde{\mathtt{f3}}\}, \epsilon\rangle\otimes f2$, indicating that `y = b.f2` is not tainted. Similarly, `z = b.f3` is determined to be untainted.

Combining two $\mathcal{F}$-strings, F_1 and F_2, through $F_1\otimes F_2$ follows Definition 4.2 straightforwardly. We can represent F_2 as $\langle L_{F_2}, K_{F_2}, S_{F_2}\rangle = \langle f_1^2 \ldots f_k^2, \{\widetilde{g_0^2}, \ldots, \widetilde{g_n^2}\}, \overline{h_1^2} \ldots \overline{h_m^2}\rangle$. The combination, $F_1\otimes F_2$, is then computed as $F_1 \otimes f_k^2 \otimes \ldots \otimes f_1^2 \otimes \widetilde{g_0^2} \otimes \ldots \otimes \widetilde{g_n^2} \otimes \overline{h_m^2} \otimes \ldots \otimes \overline{h_1^2}$, inverting L_{F_2} and S_{F_2} sequences to align with the strategy of adding new field accesses at the beginning.

4.2.3 K-Limiting. The grammar depicted in Figure 7 ensures field-sensitivity, and when combined with the context-sensitivity modeled inherently by the IDE framework [Sagiv et al. 1996], it leads to an *interleaved CFL-reachability* problem. However, this problem is known to be undecidable [Reps 2000]. To address this, similar to the approach in [Arzt et al. 2014], we make use of k-limiting, which restricts the number of symbols in a field access string. This restriction effectively transforms the context-free grammar given in Figure 7 into a regular grammar. In IDEDroid, k-limiting restricts stores and loads in a field access string to k, but allows unlimited kills while preserving decidability. With k-limiting, $F \otimes f$ treats f as ϵ if L_F hits k, while $F \otimes \overline{f}$ deletes S_F's last symbol upon reaching k.

4.3 The IDE Solver

IDEDroid uses the standard IDE solver in two phases, Phase I and Phase II, with their dynamic programming algorithms detailed in [Sagiv et al. 1996]. We briefly describe the roles of these phases, guided by the edge functions from Section 4.2.2 and illustrated in Figure 5.

In Phase I, for a node $\langle n, d \rangle$ in method p, assuming v_p is a parameter, and s_p and e_p are its unique entry and exit nodes, respectively, IDEDroid constructs a jump function from $\langle s_p, v_p \rangle$ to $\langle n, d \rangle$. This function is represented as $\langle s_p, v_p \rangle \xrightarrow{l_{\langle v_p, d \rangle}} \langle n, d \rangle$, with $l_{\langle v_p, d \rangle}$ embodying a set of field access strings. The associated transformer is $\lambda env.env[d \mapsto env(v_p) \otimes l_{\langle v_p, d \rangle}]$, with each string in $l_{\langle v_p, d \rangle}$ representing field accesses on a realizable path from $\langle s_p, v_p \rangle$ to $\langle n, d \rangle$. Additionally, IDEDroid builds function summaries based on jump functions from s_p to e_p, represented as $\langle s_p, v_p \rangle \xrightarrow{l_{\langle v_p, v'_p \rangle}} \langle e_p, v'_p \rangle$, where v'_p is a base variable. For inter-procedural call and return edges $\langle c_i, a_c \rangle \xrightarrow{\epsilon} \langle s_p, v_p \rangle$ and $\langle e_p, v'_p \rangle \xrightarrow{\epsilon} \langle r_i, a'_c \rangle$, where a'_c is a base variable, IDEDroid introduces a summary edge in the caller. This is expressed as $\langle c_i, a_c \rangle \xrightarrow{l_{\langle v_p, v'_p \rangle}} \langle r_i, a'_c \rangle$, leading to an update in the jump function for $\langle r_i, a'_c \rangle$. Phase I continues until all jump functions are computed, indicating a fixed point has been reached.

In Phase II, IDEDroid calculates the actual value for each node $\langle n, d \rangle$ by resolving $\langle s_{\mathtt{main}}, \mathbf{0} \rangle \xrightarrow{l_{\langle \mathbf{0}, d \rangle}} \langle n, d \rangle$, where d is a base variable and $l_{\langle \mathbf{0}, d \rangle}$ is a set of field access strings. Consequently, we utilize the standard $val(\langle n, d \rangle)$ function to record the set of field access strings of a base variable d at a program point n. Each string in this set is translated into a field access sequence. This sequence, combined with the base variable d, forms an access path for d. Specifically, the field access string F, which is identified as a triple $\langle L_F, K_F, S_F \rangle = \langle f_1 \ldots f_k, \{\widetilde{g_0} \ldots \widetilde{g_n}\}, \overline{h_1} \ldots \overline{h_m} \rangle$ (Definition 4.2), is converted to a field access sequence $h_1 \ldots h_m$, leading to the access path $d.h_1 \ldots h_m$. In the special case when F has no stores, the access path becomes the base variable itself d. Unbalanced loads in L_F suggest that the base variable d is loaded from unknown sources, e.g., external inputs, while K_F indicates the removal of non-relevant fields to the access path.

To optimize performance, Phase II initially determines the actual values of formal parameters through an iterative process. This process involves propagating values from actual parameters to their corresponding formal parameters and vice versa, from formal parameters back to the call sites within their containing functions. The actual value of a formal parameter is calculated by resolving $\langle s_{\mathtt{main}}, \mathbf{0} \rangle \xrightarrow{l_{\langle \mathbf{0}, v_p \rangle}} \langle s_p, v_p \rangle$, leading to $val(\langle s_p, v_p \rangle) = l_{\langle \mathbf{0}, v_p \rangle}$. Following this, the actual value $val(\langle n, d \rangle)$ for each node $\langle n, d \rangle$ in method p is determined according to the values of its parameters. This is achieved by applying the jump functions $\langle s_p, v_p^i \rangle \xrightarrow{l_{\langle v_p^i, d \rangle}} \langle n, d \rangle$ to $val(\langle s_p, v_p^i \rangle)$ for each parameter and then taking the union of these results. Here, v_p^i represents the i-th parameter (starting from 1) of method p, which has N parameters. The union is computed as $\bigcup_{i\in\{1..N\}} val(\langle s_p, v_p^i\rangle)\otimes l_{\langle v_p^i,d\rangle}$.

4.4 Revisiting the Motivating Example

Let us apply our IDE-based approach to perform taint analysis on the example in Figure 2.

4.4.1 Phase I. IDEDroid constructs jump functions by computing intra-procedural realizable paths. For example, in the then branch, three path edges: $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{f}}\}} \langle 7, \mathsf{a}\rangle$, $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{g}}\}} \langle 7, \mathsf{a}\rangle$, and $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{h}}\}} \langle 7, \mathsf{a}\rangle$, contribute to the jump function $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{f}},\overline{\mathsf{g}},\overline{\mathsf{h}}\}} \langle 7, \mathsf{a}\rangle$. Additionally, in `bar()`, the summary function $\langle 17, \mathsf{p}\rangle \xrightarrow{\{\mathsf{f}\overline{\mathsf{f}}\}} \langle 19, \mathsf{q}\rangle$ creates a summary edge $\langle 7, \mathsf{a}\rangle \xrightarrow{\{\mathsf{f}\overline{\mathsf{f}}\}} \langle 8, \mathsf{b}\rangle$ at `bar()`'s call site. Combining these functions yields $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{f}}\}} \langle 8, \mathsf{b}\rangle$ as $\{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}\otimes\{\mathsf{f}\overline{\mathsf{f}}\} = \{\overline{\mathsf{f}}\}$. This leads to the computation of $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\epsilon\}} \langle 9, \mathsf{x}\rangle$, ultimately detecting a potential leak at line 9.

In the else branch (lines 11-13), IDEDroid begins with the path edge $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{f}},\overline{\mathsf{g}},\overline{\mathsf{h}}\}} \langle 11, \mathsf{a}\rangle$. A summary function $\langle 21, \mathsf{k}\rangle \xrightarrow{\{\widetilde{\mathsf{f}}\}} \langle 22, \mathsf{k}\rangle$ then adds a summary edge $\langle 11, \mathsf{a}\rangle \xrightarrow{\{\widetilde{\mathsf{f}}\}} \langle 12, \mathsf{a}\rangle$. This leads to the jump function $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{g}},\overline{\mathsf{h}}\}} \langle 12, \mathsf{a}\rangle$, derived from $\{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}\otimes\{\widetilde{\mathsf{f}}\}$. With the edge function $\langle 12, \mathsf{a}\rangle \xrightarrow{\{\mathsf{f}\}} \langle 12, \mathsf{y}\rangle$, no viable path edge leads to $\langle 13, \mathsf{y}\rangle$ as the operation $\{\overline{\mathsf{g}}\widetilde{\mathsf{f}}, \overline{\mathsf{h}}\widetilde{\mathsf{f}}\}\otimes\{\mathsf{f}\} = \emptyset$. As a result, y is not added to D, and `sink(y)` at line 13 will not be identified as a leak.

4.4.2 Phase II. IDEDroid computes the actual values of nodes in the program's exploded graph. This involves initially determining the actual values of formal parameters by propagating values from call nodes. For example, the value for the actual parameter at node $\langle 7, \mathsf{a}\rangle$ is computed as $val(\langle 7, \mathsf{a}\rangle) = \{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}$, based on the jump function $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \xrightarrow{\{\overline{\mathsf{f}},\overline{\mathsf{g}},\overline{\mathsf{h}}\}} \langle 7, \mathsf{a}\rangle$. For the call edge $\langle 7, \mathsf{a}\rangle \rightarrow \langle 17, \mathsf{p}\rangle$, we therefore have $val(\langle 17, \mathsf{p}\rangle) = \{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}$. Similarly, for node $\langle 21, \mathsf{k}\rangle$, $val(\langle 21, \mathsf{k}\rangle) = \{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}$.

Analyzing node $\langle 18, \mathsf{v}\rangle$, we start with its jump function $\langle 17, \mathsf{p}\rangle \xrightarrow{\{\mathsf{f}\}} \langle 18, \mathsf{v}\rangle$. The value $val(\langle 18, \mathsf{v}\rangle)$ is obtained by $val(\langle 17, \mathsf{p}\rangle)\otimes\{\mathsf{f}\}$, which translates to $\{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}\otimes\{\mathsf{f}\}$, yielding $\{\epsilon\}$. This suggests that v becomes tainted before line 18. Next, considering node $\langle 22, \mathsf{k}\rangle$, its jump function is $\langle 21, \mathsf{k}\rangle \xrightarrow{\{\widetilde{\mathsf{f}}\}} \langle 22, \mathsf{k}\rangle$. Thus, $val(\langle 22, \mathsf{k}\rangle) = val(\langle 21, \mathsf{k}\rangle)\otimes\{\widetilde{\mathsf{f}}\} = \{\overline{\mathsf{f}}, \overline{\mathsf{g}}, \overline{\mathsf{h}}\}\otimes\{\widetilde{\mathsf{f}}\}$ before line 22, resulting in the set of field access strings $\{\overline{\mathsf{g}}, \overline{\mathsf{h}}\}$. These two strings indicate `k.g` and `k.h` are tainted prior to line 22.

4.5 Precision and Soundness

We discuss IDEDroid's precision and soundness, based on its field access string representation (Definition 4.2), against FlowDroid's standard access path encoding.

Lemma 1. *Without k-limiting (i.e., when $k = \infty$), for any tainted access string found by IDEDroidwithout encountering kill operations, we can always find a corresponding tainted access path in FlowDroid representing the same leak (i.e., the same leak on the same object field), and vice versa.*

Proof Sketch. Eliminating kill operations simplifies the grammar from Figure 7 to Figure 6, corresponding to the standard context-free grammar for tracking field accesses via the balanced parentheses problem [Reps 1998]. Thus, while FlowDroid explicitly models tainted data-flow facts using access paths, IDEDroid does so implicitly yet equivalently via a CFL approach. For a field access string $\langle \_, \_, \overline{h_1} \ldots \overline{h_m}\rangle$ associated with base variable v at line ℓ, IDEDroid deems $v.h1...h_m$ tainted within its CFL-reachability approach for field sensitivity (Section 4.3). Analyzing the same statement sequence from which IDEDroid derives $\langle \_, \_, \overline{h_1} \dots \overline{h_m} \rangle$, FlowDroid likewise propagates the taint $v.h1...h_m$ to line ℓ using its access-path-based method. This logic is reciprocal. □

Lemma 2. *Without k-limiting (i.e., when $k = \infty$), IDEDroid surpasses FlowDroid in precision by sanitizing a strict superset of false positives, effectively sanitizing all that FlowDroid does and more.*

Proof Sketch. Consider the scenario of taint propagation where x is already tainted at the point where $x.f = sanitize()$ occurs. Semantically, $x.f = sanitize()$ is expected to eliminate all tainted paths starting with $x.f$. Upon detecting $\widetilde{f}$, IDEDroid discards the string $\langle -, -, \overline{f} \dots \rangle$ and also uses $x.f = sanitize()$ to nullify the right-matching load $a = x.f$, turning $\langle -, \{\dots, \widetilde{f}, \dots\}, \epsilon \rangle \otimes f$ into Nil, thereby sanitizing $x.f$ even if previously tainted (Equation 1). Conversely, FlowDroid's approach is less precise; it treats all access paths from tainted x as tainted, making $x.f = sanitize()$ only cleanse $x.f$, not x, leaving a tainted after loading from still-tainted x. □

Lemma 3. *With k-limiting (Section 4.2.3), IDEDroid is strictly more precise than FlowDroid.*

Proof Sketch. In FlowDroid, a field access path with length k, denoted $v.f_1 \dots f_k$, can always be represented as a field access string with k stores in IDEDroid. IDEDroid enhances precision by simplifying matched load and store pairs to ϵ (in the fourth case of $F \otimes f$ in Equation 1), allowing it to handle access paths exceeding k in length, as shown in Figure 3. □

Figure 3 gives an example, for which IDEDroid achieves greater precision than FlowDroid under 2-limiting. FlowDroid erroneously reports a leak at line 5, but IDEDroid, using its grammar, simplifies $\langle 8, \mathsf{p} \rangle \xrightarrow{\overline{\mathsf{f2}}\ \overline{\mathsf{f1}}} \langle 9, \mathsf{x} \rangle \xrightarrow{\mathsf{f1}\ \mathsf{f2}} \langle 10, \mathsf{y} \rangle$ in `foo()` to $\langle 8, \mathsf{p} \rangle \longrightarrow \langle 10, \mathsf{y} \rangle$ (as its function summary). This leads to a *summary edge* $\langle 3, \mathsf{b1} \rangle \longrightarrow \langle 4, \mathsf{b2} \rangle$ for the call at line 3. Thus, $\langle 2, \mathbf{0} \rangle \xrightarrow{\overline{\mathsf{h}}} \langle 3, \mathsf{b1} \rangle \longrightarrow \langle 4, \mathsf{b2} \rangle \xrightarrow{\mathsf{g}} \langle 5, \mathsf{c} \rangle$ is not CFL-reachable, as the access string on that path $\overline{h}g$ is not compatible with our grammar. This leads IDEDroid to correctly infer that there are no leaks at line 5.

Theorem 1. *IDEDroid exceeds FlowDroid in precision under k-limiting with identical k values.*

Proof. This is due to IDEDroid identifying all truly tainted access paths (Lemma 1) and filtering out some falsely tainted access paths that FlowDroid does not, both without k-limiting (Lemma 2) and when k-limiting is applied (Lemma 3). □

5 Evaluation

To evaluate the impact of our research, we focused on the following research questions in our assessment of IDEDroid, which uses our CFL-based approach for field-sensitivity in the IDE framework, and FlowDroid, with its access-path-based method in the IFDS framework:

- **RQ1.** How does IDEDroid's performance compare with that of FlowDroid?
- **RQ2.** Does IDEDroid offer greater precision than FlowDroid?
- **RQ3.** Can IDEDroid facilitate tradeoffs between efficiency and precision?
- **RQ4.** Does IDEDroid maintain the known performance characteristics in Phases I and II?

Our assessment included programs on unit test cases and large, real-world Android apps.

5.1 Experimental Setting

We developed IDEDroid on top of FlowDroid, which is built using Soot [Lam et al. 2011]. Both IDEDroid and FlowDroid deploy the same demand-driven pointer analysis, Boomerang [Späth et al. 2016], for alias queries. IDEDroid generates alias queries only in Phase I when storing a tainted base variable, mirroring Boomerang's approach adopted in FlowDroid. Phase II aggregates field access strings without triggering alias queries. IDEDroid's alias handling in Phase I aligns with FlowDroid combined with Boomerang, maintaining soundness and precision as outlined in [Späth et al. 2016]. Note that FlowDroid is integrated with Boomerang, which utilizes the optimized IFDS solver, FastSolver [Arzt 2017]. In contrast, IDEDroid employs the standard Heros [Bodden 2012] IDE solver due to the absence of an optimized IDE solver at this time. In our experiments, we used the version of FlowDroid (ad6a287) from the Boomerang artifact [Späth et al. 2016], adopting its alias analysis approach for this paper. It is important to note that our IDEDroid approach is applicable to other existing IFDS-based frameworks like PhASAR [Schubert et al. 2019]. The essential component of IDEDroid, its efficient CFL-reachability solver described in Section 4.2, is comprised of approximately 800 lines of Java code and can be readily adapted to other tools.

Following prior research [Arzt 2021; Arzt et al. 2014; He et al. 2019; Lerch et al. 2015; Li et al. 2021], we imposed k-limiting (default $k = 5$) to control field access lengths and utilize FlowDroid's default sources and sinks for the apps evaluated in both tools. This practice of modifying k aims to balance performance and precision, as a higher k enhances precision at the expense of performance. Moreover, IDEDroid introduces a new balance between efficiency and precision by limiting the number of access strings per base variable, which is by default unlimited.

According to Theorem 1, our CFL-based approach outperforms the traditional access-path-based approach in precision under the same k-limiting setting. Using 327 unit tests from Boomerang [Späth et al. 2016], both IDEDroid and FlowDroid identified identical leaks, as these cases lack the complexity seen in Figure 3. This also confirms the correctness of our IDEDroid implementation. For a deeper comparison of IDEDroid's precision and efficiency against FlowDroid, we scrutinized 24 major real-world Android apps containing malware apps from Drebin [Arp et al. 2014; Spreitzenbarth et al. 2013] and benign apps from FossDroid [FossDroid 2023]. Both Drebin and FossDroid have been heavily utilized for evaluating Android malware detection techniques, as recent studies [Du et al. 2023; Yang et al. 2018, 2022] demonstrate. Initially, we selected 5 top apps from each of 17 categories in FossDroid [FossDroid 2023], which has been wildly used in previous works [He et al. 2023, 2019; Li et al. 2021], resulting in 85 apps. After excluding 53 small apps (analyzable within 2 minutes), 12 without sources or sinks, and 6 unprocessable due to odexed code, we chose 14 apps for in-depth analysis. Additionally, from Drebin [Arp et al. 2014; Spreitzenbarth et al. 2013], 100 apps were chosen randomly, including a range of diverse application categories like `DroidDream`, `Iconosys`, and `SMSreg`. Eliminating 63 small, quickly analyzed apps, 19 without sources or sinks, and 8 incompatible with Soot left us with 10 apps for detailed evaluation. We excluded those small apps where both FlowDroid and IDEDroid can complete the analysis in seconds because, for these apps, the IFDS solver contributes only a small fraction of the total analysis time, with the majority spent by Soot loading the APK prior to analysis. Consequently, the performance differences between FlowDroid and IDEDroid are marginal.

When analyzing an app, we set the timeout budget to 5 hours. All experiments were conducted on an Intel Core(TM) i5-10210U machine (2.11GHz) with 40 GB of RAM, running Ubuntu 20.04.01.

Table 1 compares IDEDroid and FlowDroid in terms of their performance, averaging results over five runs per app. Column 1 names the 24 analyzed apps (with Drebin apps anonymized), and Column 2 shows their abbreviations. Columns 3-8 detail analysis times for FlowDroid and IDEDroid. For IDEDroid in Column 4 (default setting with unrestricted number of field accesses maintained per base variable), both analysis time and speedup against FlowDroid are provided. Columns 5-8, also for IDEDroid, show analysis times and speedups (compared to its default setting) with field accesses per base variable limited to 10,000, 1,000, 100, and 10. The number of field accesses per base variable is the size of the set of field access strings ($|L|$) to which a base variable ($v \in D$) is mapped. We enforce this restriction in different settings by halting the expansion of

Table 1. Comparing IDEDroid and FlowDroid in analyzing 24 Apps. FD denotes FlowDroid. ID signifies IDEDroid, evaluated under five different settings. The default setting for IDEDroid allows unrestricted number of field accesses per base variable, with variations across 10,000, 1,000, 100, and 10 in the other settings.

APP	Abbr	Time (s)						#Leaks					
		FD	ID	10000	1000	100	10	FD	ID	10000	1000	100	10
com.mschlauch.comfortreader	CMC	>5h	2,751.6 6.5×	2,045.4 1.3×	440.3 6.2×	258.5 10.6×	255.3 10.8×	-	25	25	25	21	19
dk.jens.backup	DJB	>5h	41.3 435.8×	43.6 0.9×	39.2 1.1×	32.4 1.3×	24.5 1.7×	-	48	48	48	45	45
org.jfedor.frozenbubble	OJF	OOM	OOM	OOM	3,714.7	1,709.2	1,729.3	-	-	-	93	93	93
deep.ryd.rydplayer	DRR	>5h	26.4 681.8×	25.7 1.0×	25.7 1.0×	25.8 1.0×	26.4 1.0×	-	4	4	4	4	4
org.icasdri.mather	OIM	OOM	>5h	>5h	>5h	5,584.1	3,949.1	-	-	-	-	1	1
org.billthefarmer.editor	OBE	>5h	1,755.9 10.3×	1,533.4 1.1×	891.3 2.0×	740.5 2.4×	924.8 1.9×	-	11	11	11	10	10
protect.babymonitor	PB	>5h	90.8 198.2×	86.5 1.1×	84.8 1.1×	130.1 0.7×	4.2 21.8×	-	1	1	1	1	1
com.fr3ts0n.ecu.gui.androbd	CFEGA	>5h	34.0 529.4×	32.0 1.1×	30.3 1.1×	31.0 1.1×	32.5 1.0×	-	10	10	10	9	9
com.google.zxing.client.android	CGZCA	12,741.0	425.8 29.9×	416.3 1.0×	399.1 1.1×	430.7 1.0×	369.7 1.2×	26	23	23	23	23	23
org.krita	OK	4,312.3	73.0 59.1×	65.9 1.1×	72.0 1.0×	70.6 1.0×	70.4 1.0×	5	4	4	4	4	4
org.michaelevans.nightmodeenabler	OMN	1,165.6	430.4 2.7×	422.7 1.0×	448.3 1.0×	420.2 1.0×	426.4 1.0×	1	1	1	1	1	1
org.dnaq.dialer2	ODD	336.8	4.2 80.2×	4.0 1.0×	4.2 1.0×	4.1 1.0×	4.1 1.0×	25	25	25	25	25	25
net.ivpn.client	NIC	212.4	94.3 2.3×	89.2 1.1×	96.9 1.0×	91.8 1.0×	99.4 0.9×	7	7	7	7	7	7
fr.neamar.kiss	FNK	122.3	59.2 2.1×	58.7 1.0×	59.5 1.0×	58.4 1.0×	49.6 1.2×	13	13	13	13	13	13
DroidDream-76e8	DroidDream	>5h	208.8 86.2×	164.6 1.3×	155.8 1.3×	161.1 1.3×	159.4 1.3×	-	28	28	28	28	28
Iconosys-15ae	Iconosys	>5h	82.3 218.8×	90.9 0.9×	87.4 0.9×	96.5 0.9×	84.0 1.0×	-	76	76	76	78	76
Plankton-0320	Plankton	>5h	1,377.8 13.1×	1,303.4 1.1×	1,255.9 1.1×	1,258.8 1.1×	1,256.6 1.1×	-	151	151	147	147	147
SMSreg-46e4	SMSreg	>5h	7.6 2, 368.4×	7.7 1.0×	7.7 1.0×	6.9 1.1×	6.4 1.2×	-	15	15	15	15	15
SendPay-99ed	SendPay	7,012.3	246.3 28.5×	241.0 1.0×	154.4 1.6×	124.7 2.0×	104.9 2.3×	120	120	121	119	120	95
Adrd-9f73	Adrd	3,067.0	240.9 12.7×	181.7 1.3×	154.5 1.6×	130.0 1.9×	157.3 1.5×	6	6	6	6	6	6
DroidKungFu-191b	DroidKungFu	1,280.0	19.1 67.0×	18.6 1.0×	18.2 1.0×	18.3 1.0×	19.6 1.0×	3	3	3	3	3	3
BaseBridge-04ef	BaseBridge	1,186.6	34.0 34.9×	35.2 1.0×	40.4 0.8×	49.8 0.7×	42.4 0.8×	13	13	13	13	13	13
Geinimi-6902	Geinimi	509.0	48.0 10.6×	42.4 1.1×	53.1 0.9×	36.4 1.3×	48.3 1.0×	21	18	18	18	18	18
FakeDoc-b3a8	FakeDoc	147.9	23.1 6.4×	23.5 1.0×	24.2 1.0×	22.0 1.1×	21.1 1.1×	26	25	25	25	25	25

lattice L at distinct set limits. The last six columns record leaks detected by both tools under all settings, highlighting their leak detection capabilities.

5.2 RQ1: Efficiency Improvement

IDEDroid significantly outperforms FlowDroid in terms of efficiency and also consumes less memory, as detailed in Table 1 and Figure 9. We initially focus on the efficiency and memory usage benefits of IDEDroid, followed by an analysis of the underlying reasons for these advantages.

5.2.1 Speedups. In the case of OJF and OIM, both FlowDroid and IDEDroid (in its default setting) failed to complete their analysis due to memory constraints or exceeding the 5-hour time limit. However, for the remaining 22 apps, as indicated in Columns 3-4 of Table 1, IDEDroid significantly outperforms FlowDroid, achieving speedups ranging from 2.1× (FNK) to an impressive 2, 368.4× (SMSreg), with an average speedup of 222.0×. It is worth emphasizing that IDEDroid was able to analyze 10 more apps than FlowDroid (e.g., CMC and DJB) within the 5-hour limit.

For the 10 apps that FlowDroid could not analyze, we conservatively estimated its analysis time at 5 hours each, implying IDEDroid's actual speedups could be higher. Remarkably, IDEDroid analyzed SMSreg in only 7.6 seconds, a task that FlowDroid could not complete in 5 hours.

Apps with moderate speedups, such as FNK (2.1x) and NIC (2.3x), had shorter analysis times with IDEDroid, completing in 59.2 and 94.3 seconds respectively.

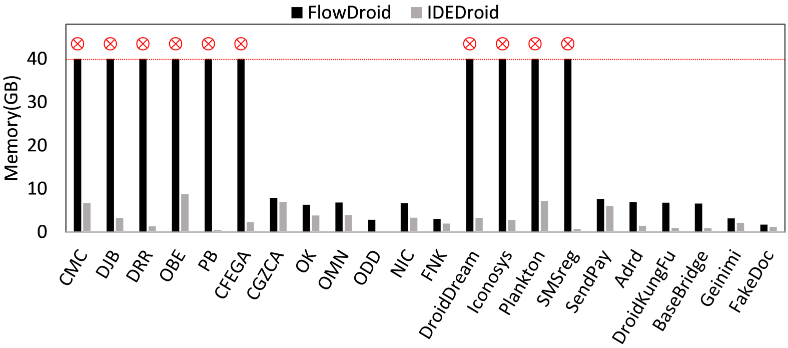


Fig. 9. Comparing IDEDroid and FlowDroid (in its default setting) in terms of memory consumed. ⊗ indicates apps where FlowDroid either timed out or exhausted memory resources.

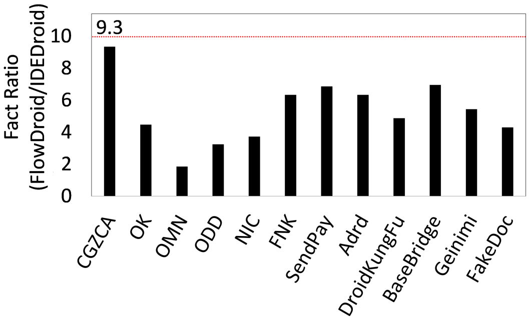


Fig. 10. The ratio of the data-flow facts processed by FlowDroid over that of IDEDroid.

5.2.2 Memory Usage. Figure 9 illustrates the memory consumption comparison between IDEDroid (in its default setting) and FlowDroid, excluding the two apps, OJF and OIM, where both tools failed to complete within the 5-hour time frame. Consistently, IDEDroid demonstrated lower memory usage than FlowDroid. The ratio of FlowDroid's memory consumption to that of IDEDroid varied, ranging from 1.1× for CGZCA to as high as 7.4× for BaseBridge, averaging at about 3.6×.

5.2.3 Analysis. In Section 2.2, we highlighted three key benefits of our CFL-based approach over the traditional access-path-based approach for enforcing field-sensitivity in the IFDS/IDE framework: (1) a smaller data-flow fact domain, (2) less propagation of data-flow facts, and (3) fewer alias queries. We will now provide experimental evidence demonstrating how these benefits lead to the observed enhancements in IDEDroid's performance and memory usage.

Data-flow Fact Domain. As discussed in Section 1, the time and space complexities of alias-aware IFDS/IDE algorithms are mainly driven by the data-flow facts processed. FlowDroid represents these facts using explicit access paths, whereas IDEDroid focuses solely on base variables. For the 12 apps that FlowDroid could analyze, Figure 10 shows that, on average, FlowDroid processes 5.3× more facts than IDEDroid, peaking at 9.3× for CGZCA.

Taint Propagation. In the IFDS/IDE framework, data-flow fact propagation is typically measured by the total path and summary edges processed. For the 12 apps that FlowDroid could analyze, as Table 2 illustrates, FlowDroid processed considerably more edges than IDEDroid: 10.8× more path edges (with a maximum of 18.1×) and 18.1× more summary edges (reaching up to 30.9×).

IDEDroid's CFL-based approach combines field accesses of a base variable for joint propagation in jump function calculations, thereby minimizing taint propagation. Conversely, FlowDroid's

Table 2. Comparing IDEDroid and FlowDroid on the number of path edges, summary edges and the number of alias queries processed. FD denotes FlowDroid while ID denotes IDEDroid.

APP	# Path Edges			# Summary Edges			# Alias Queries		
	FD	ID		FD	ID		FD	ID	
CGZCA	2,089,762	149,864	13.9×	100,900	4,555	22.2×	3,003	940	3.2×
OK	169,349	41,936	4.0×	5,583	1,139	4.9×	1,296	276	4.7×
OMN	6,223	69,201	0.1×	435	866	0.5×	130	116	1.1×
ODD	120,237	7,992	15.0×	23,364	757	30.9×	690	97	7.1×
NIC	199,452	28,428	7.0×	10,487	970	10.8×	457	183	2.5×
FNK	158,504	30,603	5.2×	9,804	1,301	7.5×	513	272	1.9×
Adrd	88,151	4,879	18.1×	4,475	339	13.2×	1,033	25	41.3×
SendPay	5,535,378	1,824,110	3.0×	26,535	6,801	3.9×	3,722	1,271	2.9×
DroidKungFu	37,460	5,139	7.3×	1,685	224	7.5×	454	49	9.3×
BaseBridge	54,432	21,243	2.6×	3,006	1,397	2.2×	537	114	4.7×
Geinimi	125,738	62,622	2.0×	6,493	2,424	2.7×	455	220	2.1×
FakeDoc	696,315	140,061	5.0×	13,080	2,601	5.0×	1174	626	1.9×
Average	773,417	198,840	10.8×	17,154	1,948	18.1×	1,122	349	10.1×

access-path approach results in more path and summary edges. The application of k-limiting to access paths in FlowDroid also contributes to unnecessary propagation of spuriously tainted access paths. IDEDroid addresses this limitation by applying CFL-reachability in function summary computations (as motivated in Section 3.3), which substantially reduces these edges, improves performance relative to FlowDroid, and enhances precision, as will be described in Section 5.3.

Alias Queries. In the access-path-based approach, each access path newly tainted at the same field store triggers a separate alias query, even when they share the same base variable, leading to numerous redundant queries. Conversely, IDEDroid's CFL-based approach, by compactly encoding multiple access paths for a single base variable, generates just one alias query per store. As a result, FlowDroid ends up issuing 10.1× more alias queries than IDEDroid, reaching 41.3× for Adrd.

5.3 RQ2: Precision Improvement

Theoretically, IDEDroid surpasses FlowDroid in precision, as delineated in Theorem 1. In practical scenarios, IDEDroid demonstrates enhanced precision over FlowDroid for certain apps. Among the 12 apps FlowDroid managed to analyze within a 5-hour frame (Table 1), IDEDroid exhibited notable precision improvements in four apps: OK by 20.0%, Geinimi by 14.3%, CGZCA by 11.5%, and FakeDoc by 3.8%. For the remaining 8 apps, both tools identified the same leaks, culminating in an average precision uplift of 4.1% for the 12 apps.

IDEDroid has effectively eliminated a total of eight false positives reported by FlowDroid: 1 in OK, 3 in Geinimi, 3 in CGZCA, and 1 in FakeDoc. The last ten apps mentioned in Table 1 from Drebin had their source code unavailable, necessitating a manual examination of their Jimple IR. In Geinimi, IDEDroid eliminated three false positives in the run() method of class com.admob.android.ads.AdView$a and an obfuscated method b() in class com.admob.android.ads.AdManager. These leaks trace back to a single callback source, the setBackgroundColor(int) method in class com.admob.android.ads.AdView. The related sinks, v(String,String) (appearing twice), and d(String,String), are located in the class android.util.Log. In FakeDoc, IDEDroid removed a false positive flowing from a source to a sink in onCreate(android.os.Bundle) of class com.itframework.installer.util.InstallNonMarketFromUrlActivity. For OK, IDEDroid avoided one false positive in the startApp(boolean) method of class org.qtproject.qt5.android.bindings.QtLoader from a callback source. For CGZCA, IDEDroid eliminated three false positives in the initFromCameraParameters() method of class com.google.zxing.client.

Table 3. Comparing IDEDroid and FlowDroid with k ranging from 1 to 4 under k-limiting. FD denotes FlowDroid while ID denotes IDEDroid.

APP	Time (s)								#Leaks							
	k=1		k=2		k=3		k=4		k=1		k=2		k=3		k=4	
	FD	ID	FD	ID	FD	ID	FD	ID	FD	ID	FD	ID	FD	ID	FD	ID
`CGZCA`	1443.4	229.8 6.3×	1741.3	237.3 7.3×	3223.1	282.3 11.4×	7885.1	275.8 28.6×	48	46	31	24	26	23	26	23
`OK`	828.3	71.6 11.6×	1373.1	72.4 19.0×	2099.9	78.7 26.7×	3319.9	71.2 46.6×	5	4	5	4	5	4	5	4
`OMN`	1529.5	447.8 3.4×	1538.3	460.8 3.3×	924.7	450.4 2.1×	1120.8	432.9 2.6×	1	1	1	1	1	1	1	1
`ODD`	32.6	4.2 7.8×	142	4.2 33.8×	240.9	4.0 60.2×	264.3	4.3 61.5×	25	25	25	25	25	25	25	25
`NIC`	253.1	134.2 1.9×	217.3	104.7 2.1×	202.7	107.6 1.9×	208.6	116.5 1.8×	11	10	7	7	7	7	7	7
`FNK`	90.6	80.1 1.1×	132.9	59.5 2.2×	126.7	71.6 1.8×	135.6	57.4 2.4×	13	13	13	13	13	13	13	13
`SendPay`	178.9	124.9 1.4×	240.1	123.8 1.9×	480	117.6 4.1×	1093.5	142.4 7.7×	123	123	120	120	120	120	120	120
`Adrd`	4614.7	777.4 5.9×	3708.7	1496.6 2.5×	>5h	151.2 119.0×	5121	499.9 10.2×	6	6	6	6	-	6	6	6
`DroidKungFu`	148.8	18.1 8.2×	86.8	19.2 4.5×	208.6	19.0 11.0×	594.1	17.8 33.4×	3	3	3	3	3	3	3	3
`BaseBridge`	523.7	38 13.8×	1016.5	34.3 29.6×	1463.1	33.9 43.2×	1275.1	35.8 35.6×	14	14	13	13	13	13	13	13
`Geinimi`	590.73	35.9 16.5×	418.8	73.3 5.7×	265.8	41.2 6.5×	649.7	48.1 13.5×	25	23	21	18	21	18	21	18
`FakeDoc`	40.1	21.5 1.9×	56.1	23.1 2.4×	139.5	23.6 5.9×	155.5	27.1 5.7×	26	25	26	25	26	25	26	25

`android.camera.CameraConfigurationManager` from three separate sources: `onResume()` in class `com.google.zxing.client.android.CaptureActivity`, `buildHistoryItems()` in class `com.google.zxing.client.android.history.HistoryManager`, and `buildHistoryItem(int)` in class `com.google.zxing.client.android.history.HistoryManager`.

In industry settings, reducing false positives even marginally is crucial. Such improvements in taint analysis tools like IDEDroid can significantly enhance software development efficiency, allowing developers to focus on genuine issues rather than spending time on non-existent problems.

The precision enhancements observed in four apps are attributed to a distinct code pattern illustrated in Figure 3. Additionally, in the pattern "`x = source(); x.f = sanitize(); y = x.f; sink(y)`", FlowDroid and tools using a similar access-path-based method will mistakenly report a false positive at the sink because they mark `x.f`* as tainted right from the source. In contrast, IDEDroid avoids this false positive by precisely tracking that only ϵ is carried along the path and correctly identifying the sanitization at `x.f`. Yet, it is important to note that IDEDroid has not dismissed any false positives reported by FlowDroid for this specific pattern in our evaluation.

As discussed in Sections 3.3 and 4.2, IDEDroid employs CFL-reachability for function summary computations. This approach circumvents the imprecision typical of FlowDroid's access-path-based abstraction, thus highlighting IDEDroid's enhanced precision (along with its efficiency) over FlowDroid, under k-limiting using the same k value.

5.4 RQ3: Efficiency and Precision Tradeoffs

IDEDroid offers two strategies for balancing precision and efficiency: k-limiting to cap field access lengths and limiting the number of field access strings per base variable.

Limiting Field Access Lengths. Applying k-limiting, FlowDroid caps access path lengths at k, whereas IDEDroid equivalently limits store and load labels in a field access string to k (Section 4.2.3). Table 3 shows performance and precision comparisons between both tools under varying k-limiting

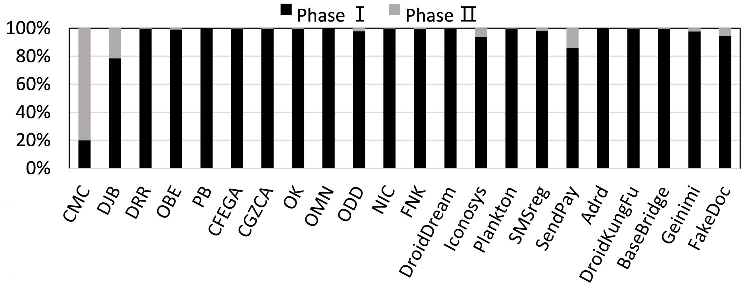


Fig. 11. The time spent by Phase I against the time spent by Phase II in IDEDroid.

values ($k = 1, 2, 3, 4$), focusing on 12 apps FlowDroid scaled with under 5-limiting (Table 1). In all scenarios, IDEDroid surpasses FlowDroid. With $k = 1$, speedups range from 1.1× (FNK) to 16.5× (Geinimi), averaging 6.6×. At $k = 2$, speedups vary from 1.9× (SendPay) to 33.8× (ODD), with an average of 9.5×. For $k = 3$ and $k = 4$, improvements continue, with speedups between 1.8× (FNK) to 119.0× (Adrd) and 1.8× (NIC) to 61.5× (ODD) respectively, and average speedups of 24.5× and 20.8×.

Precision-wise, IDEDroid matches or exceeds FlowDroid across all k values, with precision naturally enhancing as k grows for both FlowDroid and IDEDroid. However, the IFDS performance varies widely with k, affected by changes in the number of access paths maintained and propagated. For CGZCA, OK, ODD, SendPay, and FakeDoc, FlowDroid performs significantly better with k increases, not applying to the other seven apps. Specifically, FlowDroid struggles under 3-limiting with Adrd, unable to finish within 5 hours, a contrast to its capability at $k \in \{1, 2, 4\}$ due to the cost of managing numerous, often unnecessary, access paths. Conversely, IDEDroid's performance remains stable across k variations, as its base variable domain remains largely unchanged with different k settings.

Limiting Field Accesses Per Base Variable. IDEDroid offers an alternative strategy for enhancing efficiency and precision by capping field accesses per base variable, typically unrestricted. This aspect is a key feature of our approach, achieved by halting the expansion of lattice L in our IDE formulation at a set limit, as described in Section 4.2.3. Practically, this limit is seldom reached for most base variables. Although this could marginally affect soundness by not tracking every field access, it significantly boosts performance with minimal impact on leak detection capabilities.

We evaluated four variants of IDEDroid- IDEDroid$_{10000}$, IDEDroid$_{1000}$, IDEDroid$_{100}$, and IDEDroid$_{10}$, limiting field accesses per base variable to 10,000, 1,000, 100, and 10, respectively. Their efficiency and precision are provided in Columns 5-8 and 11-14 of Table 1, using IDEDroid as the baseline for computing their speedups. IDEDroid$_{1000}$ analyzed one additional app, OJF, in 3,714.7 seconds, which IDEDroid could not. IDEDroid$_{100}$ managed to analyze both OJF and OIM (which IDEDroid failed) in 1,709.2 and 5,584.1 seconds, respectively. The precision for most apps (16 out of 22 that IDEDroid could analyze) remains unchanged even for IDEDroid$_{100}$ and IDEDroid$_{10}$. For these 22 apps, average speedups of IDEDroid$_{10000}$, IDEDroid$_{1000}$, IDEDroid$_{100}$, and IDEDroid$_{10}$ over IDEDroid are 1.1×, 1.5×, 1.9×, and 2.8×, respectively, with maximum speedups being 1.3× (CMC), 6.2× (CMC), 10.6× (CMC), and 21.8× (PB) as shown in Columns 5-8 of Table 1.

Let us explore the efficiency and precision tradeoffs shown by different versions of IDEDroid. In most Android apps, classes usually possess few fields, translating into a small number of field accesses for each base variable. This is further diminished by k-limiting. As a result, in apps where the data-flow facts excluded due to the cap on the number of field accesses per base variable are minimal, the impact on IDEDroid's performance and precision is often negligible.

5.5 RQ4: Performance of IDEDroid's Two Phases

Like other IDE-based analyses, IDEDroid operates in two phases: Phase I and Phase II. For taint analysis, IDEDroid symbolically computes field accesses in Phase I by calculating jump functions and then actualizes these field accesses as access paths in Phase II, which can be resource-intensive. We assessed the efficiency of both phases, with Figure 11 displaying their percentage contributions to IDEDroid's total analysis time for the 22 apps IDEDroid could analyze, averaging at 93.7% for Phase I and 6.3% for Phase II. Generally, Phase II takes considerably less time compared to Phase I, with it finishing in under 10 seconds for 19 apps. This underscores that Phase I is typically the more complex phase in IDE-based analysis, as noted in [Sagiv et al. 1996]. However, Phase II can be demanding in specific cases, such as accounting for 80.1% of the analysis time for CMC.

Importantly, IDEDroid optimizes Phase I by focusing on base variables, significantly reducing unnecessary access path propagation. This is crucial for IDEDroid's improved performance compared to FlowDroid, illustrating that the additional Phase II introduces only a slight overhead.

6 Related Work

In this section, we review the previous works that are closely related to this research.

6.1 Field Sensitivity Models

Field-sensitivity is vital for precise static program analysis, primarily addressed through storeless and store-based models [Kanvar and Khedker 2016]. Storeless models perceive the heap as access paths comprising a base variable, followed by some field accesses. Unchecked, these paths risk algorithm non-termination. To counter this, size-limiting strategies have been developed. k-limiting [Arzt et al. 2014; De and D'Souza 2012; He et al. 2019; Landi and Ryder 1992; Tripp et al. 2013] controls the number of field accesses per access path. Regular-expression-based methods [Deutsch 1994; Khedker et al. 2007; Lerch et al. 2015; Matosevic and Abdelrahman 2012] replace field accesses with regular expressions. Logic-based approaches [Bozga et al. 2004; Kuncak et al. 2006; Lam et al. 2005; Møller and Schwartzbach 2001] utilize logical expressions. These solutions provide different levels of abstraction to maintain analysis precision and practicality.

6.2 The IFDS/IDE Framework

The IFDS/IDE framework, widely used in program optimization, verification, and security, was pioneered by [Reps et al. 1995] for solving inter-procedural, finite, subset problems and extended by Sagiv et al. [1996] to tackle inter-procedural distributed environment (IDE) issues using symbol-to-value data-flow mappings. Enhancements by Naeem et al. [2010] have been adopted in popular frameworks like WALA [IBM 2006], Soot [Lam et al. 2011], and LLVM [Lattner and Adve 2004].

Several orthogonal optimization techniques have been developed to enhance IFDS/IDE performance. Bodden [2012] developed a multi-threaded solver in Soot, and Schubert et al. [2019] created a specialized solver for C/C++ in LLVM. He et al. [2019] introduced an optimization using sparse propagation of data-flow facts, improving performance and memory efficiency. Li et al. [2021] enhanced IFDS scalability by removing inaccessible path edges and managing memory usage. Arzt [2021] and He et al. [2023] focused on efficient garbage collection methods. Wang et al. [2023] proposed a parallel IFDS algorithm with a streaming-based, out-of-core computation, while Gui et al. [2023] optimized IFDS-based taint analysis by merging equivalent value flows. Conrado et al. [2023] accelerated Phase II in IDE by exploiting structural sparsity properties of control-flow graphs and call graphs to enhance querying performance from a source point to a sink point. This approach can be adopted to further improve the performance of Phase II in IDEDroid.

IFDS-APA [Lerch et al. 2015] boosts efficiency by bundling access paths with identical base variables, which are later disaggregated for precision, achieving a modest average speedup of 2.1× over FlowDroid under 5-limiting [Lerch et al. 2015][Table 4.3]. In contrast, our IDEDroid approach achieves an average speedup of 222.0×. IDEDroid was not compared with IFDS-APA due to its absence in FlowDroid or Heros [Bodden 2012] and the lack of an accessible implementation.

Taint analysis, crucial for detecting unauthorized information flows, is a key component in various security assessment tools. In Android security analysis, numerous tools have been developed [Arzt et al. 2014; Cai and Jenkins 2018; Gordon et al. 2015; Klieber et al. 2014; Li et al. 2015; Wei et al. 2014], with FlowDroid standing out as a leading solution [Pauck et al. 2018]. This paper introduces a CFL-based heap abstraction for field-sensitivity, seamlessly integrated into an alias-aware IFDS/IDE framework through encoding in IDE's edge functions. We have implemented this abstraction in IDEDroid, a new taint analysis tool, which significantly enhances the performance of FlowDroid.

6.3 CFL-Reachability

Originally introduced for database theory [Yannakakis 1990], CFL-reachability has become a key framework for various program analysis issues [Reps 1998], such as pointer analysis [Sridharan and Bodík 2006; Sridharan et al. 2005; Zheng and Rugina 2008], program slicing [Reps et al. 1994; Sridharan et al. 2007], data-flow analysis [Arzt et al. 2014], and shape analysis [Reps 1998]. The IFDS/IDE framework, in fact, is an implementation of CFL-reachability, modeling context sensitivity via extended Dyck-CFL reachability [Kodumal and Aiken 2004; Shi et al. 2022].

CFL-reachability has seen extensive advancements in recent years, including transitive redundancy elimination [Lei et al. 2022], multi-derivation [Shi et al. 2023, 2024], offline graph simplification [Lei et al. 2023; Li et al. 2022b], disk-based computation methods [Wang et al. 2017], and efficient algorithms for (Bidirected) Dyck-CFL reachability [Chatterjee et al. 2017; Li et al. 2022a; Zhang et al. 2013]. Our research contributes to this field by framing field sensitivity as a CFL-reachability problem (involving partially matched parentheses) and addressing it during jump function calculations in the IDE framework.

Our study relates to the *interleaved CFL-reachability* issue, where multiple CFL-reachability cases jointly model program behavior. Field- and context-sensitivity in CFL-based analyses are often depicted by two interleaved CFLs. However, fully integrating these sensitivities in data-flow analysis is undecidable [Reps 2000], leading to the development of practical approximation techniques.

Recent progress includes Zhang and Su's work [Zhang and Su 2017] using *linear conjunctive language* (LCL) for interleaved Dyck-CFL problems and a corresponding LCL-reachability approximation algorithm. The concept of *synchronized pushdown systems* (SPDS) [Späth et al. 2019] also plays a role, combining results from two separate pushdown systems for reachability analysis.

Our work contributes a novel angle by elevating field-sensitive IFDS-based data-flow analysis to an IDE problem. Here, context-sensitivity is naturally embedded within the IDE algorithm, and field-sensitivity is integrated through a CFL, employing k-limiting to maintain decidability. Meantime, existing CFL-based alias analyses [Lu and Xue 2019; Shang et al. 2012; Sridharan and Bodík 2006; Xu et al. 2009; Zheng and Rugina 2008] are context- and field-sensitive but flow-insensitive, utilizing a pointer assignment graph as the underlying labeled graph. In contrast, our CFL-reachability analysis operates within an IFDS-based taint analysis framework and introduces two significant differences: (1) our CFL formulation is not only fully context- and field-sensitive but also flow-sensitive and considers kill operations, and (2) it employs CFL-based transformers with linear or constant time complexity at each step to propagate tainted base variables, while traditional CFL analyses iteratively compute reachable paths until a fixed point is achieved.

7 Conclusion

We have developed a novel field-sensitive approach that reconceptualizes access path creation into a CFL and frames it as an IDE problem. By modeling field accesses through a CFL-reachability approach, resolved during jump function computation within the IDE framework, we achieve both performance and precision improvements over the traditional access-path-based approach.

Performance-wise, our technique significantly decreases the domain of data-flow facts, leading to fewer path edges, summary edges processed, and alias queries. As a result, IDEDroid, leveraging our CFL-based approach for field-sensitivity, demonstrates superior performance and improved precision compared to FlowDroid, which uses access paths for field-sensitivity.

Data-Availability Statement

The artifact is publicly available at [Li 2024].

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (62402474, 62132020 and 62202452), and the China Postdoctoral Science Foundation (2024M753295).

References

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.* The Internet Society, San Diego, California, 23–26. https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket

Steven Arzt. 2017. *Static Data Flow Analysis for Android Applications.* Ph. D. Dissertation. Darmstadt University of Technology, Germany. http://tuprints.ulb.tu-darmstadt.de/5937/

Steven Arzt. 2021. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21).* IEEE Press, Madrid, Spain, 1098–1110. https://doi.org/10.1109/ICSE43902.2021.00102

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14).* Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15).* IEEE Press, Florence, Italy, 426–436. https://doi.org/10.1109/ICSE.2015.61

Eric Bodden. 2012. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (Beijing, China) *(SOAP '12).* Association for Computing Machinery, New York, NY, USA, 3–8. https://doi.org/10.1145/2259051.2259052

Marius Bozga, Radu Iosif, and Yassine Lakhnech. 2004. On Logics of Aliasing. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, Verona, Italy, 344–360. https://doi.org/10.1007/978-3-540-27864-1_25

Haipeng Cai and John Jenkins. 2018. Leveraging Historical Versions of Android Apps for Efficient and Precise Taint Analysis. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18).* Association for Computing Machinery, New York, NY, USA, 265–269. https://doi.org/10.1145/3196398.3196433

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/10.1145/3158118

Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun Chen Tsai, and Ahmed Khaled Zaher. 2023. Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1993–2022. https://doi.org/10.1145/3622868

Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 665–687. https://doi.org/10.1007/978-3-642-31057-7_29

Alain Deutsch. 1994. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM Sigplan Notices* 29, 6 (1994), 230–241. https://doi.org/10.1145/773473.178263

Xuechao Du, Xiang Pan, Yinzhi Cao, Boyuan He, Gan Fan, Yan Chen, and Daigang Xu. 2023. FlowCog: Context-Aware Semantic Extraction and Analysis of Information Flow Leaks in Android Apps. *IEEE Transactions on Mobile Computing* 22, 11 (2023), 6460–6476. https://doi.org/10.1109/TMC.2022.3197638

FossDroid 2023. *Fossdroid: Free and open source Android apps.* Retrieved 2023 from https://fossdroid.com/

Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, Vol. 15. The Internet Society, San Diego, California, 110. https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

Alexey Gotsman, Josh Berdine, and Byron Cook. 2006. Interprocedural Shape Analysis with Separated Heap Abstractions. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4134)*, Kwangkeun Yi (Ed.). Springer, Seoul, Korea, 240–260. https://doi.org/10.1007/11823230_16

Yujiang Gui, Dongjie He, and Jingling Xue. 2023. Merge-Replay: Efficient IFDS-Based Taint Analysis by Consolidating Equivalent Value Flows. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE56229.2023.00027

Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 69–82. https://doi.org/10.1145/512529.512539

Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. 2023. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses Using Fine-Grained Garbage Collection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3597926.3598041

Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, San Diego, California, 267–279. https://doi.org/10.1109/ASE.2019.00034

IBM 2006. *WALA: T.J. Watson Libraries for Analysis.* Retrieved 2023 from http://wala.sourceforge.net

Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–47. https://doi.org/10.1145/2931098

Uday P Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 1–es. https://doi.org/10.1145/1290520.1290521

William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614633

John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. *ACM Sigplan Notices* 39, 6 (2004), 207–218. https://doi.org/10.1145/996841.996867

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin C Rinard. 2006. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering* 32, 12 (2006), 988–1005. https://doi.org/10.1109/TSE.2006.125

Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. http://www.bodden.de/pubs/lblh11soot.pdf

Patrick Lam, Viktor Kuncak, and Martin Rinard. 2005. Generalized Typestate Checking for Data Structure Consistency. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation* (Paris, France) *(VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 430–447. https://doi.org/10.1007/978-3-540-30579-8_28

William Landi and Barbara G Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices* 27, 7 (1992), 235–248. https://doi.org/10.1145/143095.143137

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime*

Optimization (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1556–1582. https://doi.org/10.1145/3563343

Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023. Recursive State Machine Guided Graph Folding for Context-Free Language Reachability. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 318–342. https://doi.org/10.1145/3591233

Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, Lincoln, Nebraska, 619–629. https://doi.org/10.1109/ASE.2015.9

Haofeng Li. 2024. Boosting the Performance of Alias-Aware IFDS Analysis with CFL-based Environment Transformers (Artifact). (7 2024). https://doi.org/10.6084/m9.figshare.26105056.v1

Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Lian Li, and Lin Gao. 2024. Boosting the Performance of Multi-Solver IFDS Algorithms with Flow-Sensitivity Optimizations. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 296–307. https://doi.org/10.1109/CGO57630.2024.10444884

Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling up the IFDS Algorithm with Efficient Disk-Assisted Computing. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, Virtual Event, Republic of Korea, 236–247. https://doi.org/10.1109/CGO51591.2021.9370311

Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Florence, Italy, 280–291. https://doi.org/10.1109/ICSE.2015.48

Yuanbo Li, Kris Satya, and Qirun Zhang. 2022a. Efficient algorithms for dynamic bidirected Dyck-reachability. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498724

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2022b. Fast Graph Simplification for Interleaved-Dyck Reachability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–28. https://doi.org/10.1145/3492428

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360574

Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA) *(SIGSOFT '04/FSE-12)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/1029894.1029907

Ivan Matosevic and Tarek S. Abdelrahman. 2012. Efficient Bottom-up Heap Analysis for Symbolic Path-Based Data Access Summaries. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 252–263. https://doi.org/10.1145/2259016.2259049

Anders Møller and Michael I Schwartzbach. 2001. The pointer assertion logic engine. *ACM SIGPLAN Notices* 36, 5 (2001), 221–231. https://doi.org/10.1145/378795.378851

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144. https://doi.org/10.1007/978-3-642-11970-5_8

Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19, 5 (1994), 11–20. https://doi.org/10.1145/193173.195287

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131 – 170. https://doi.org/10.1016/0304-3975(96)00072-2

Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe (Eds.). ACM, 264–274. https://doi.org/10.1145/2259016.2259050

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2023. Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 624–636. https://doi.org/10.1109/ASE56229.2023.00118

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2024. PEARL: A Multi-Derivation Approach to Efficient CFL-Reachability Solving. *IEEE Transactions on Software Engineering* (2024), 1–19. https://doi.org/10.1109/TSE.2024.3437684

Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the extended Dyck-CFL reachability for context-sensitive program analysis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1438–1468. https://doi.org/10.1145/3563339

Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and Precise Alias-Aware Dataflow Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 99 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133923

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Rome, Italy, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java (Artifact). *Dagstuhl Artifacts Series* 2, 1 (2016), 12:1–12:2. https://doi.org/10.4230/DARTS.2.1.12

Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (Coimbra, Portugal) *(SAC '13)*. Association for Computing Machinery, New York, NY, USA, 1808–1815. https://doi.org/10.1145/2480362.2480701

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*. 112–122. https://doi.org/10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices* 40, 10 (2005), 59–76. https://doi.org/10.1145/1103845.1094817

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7793)*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer, Rome, Italy, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. *SIGPLAN Not.* 44, 6 (June 2009), 87–97. https://doi.org/10.1145/1543135.1542486

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404. https://doi.org/10.1145/3093336.3037744

Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. 2023. DStream: A Streaming-Based Highly Parallel IFDS Framework. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2488–2500. https://doi.org/10.1109/ICSE48619.2023.00208

Gary Wassermann and Zhendong Su. 2008. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 171–180. https://doi.org/10.1145/1368088.1368112

Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP*, Vol. 9. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

Wei Yang, Mukul Prasad, and Tao Xie. 2018. EnMobile: Entity-Based Characterization and Analysis of Mobile Malware. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 384–394. https://doi.org/10.1145/3180155.3180223

Zhi Yang, Zhanhui Yuan, Shuyuan Jin, Xingyuan Chen, Lei Sun, Xuehui Du, Wenfa Li, and Hongqi Zhang. 2022. FSAFlow: Lightweight and Fast Dynamic Path Tracking and Control for Privacy Protection on Android Using Hybrid Analysis with State-Reduction Strategy. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2114–2129. https://doi.org/10.1109/SP46214.2022.9833764

Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 230–242. https://doi.org/10.1145/298514.298576

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 435–446. https://doi.org/10.1145/2491956.2462159

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 344–358. https://doi.org/10.1145/3093333.3009848

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208. https://doi.org/10.1145/1328897.1328464

Received 2024-04-06; accepted 2024-08-18