Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis

Dongjie He*†‡, Haofeng Li†‡, Lei Wang†‡, Haining Meng†‡,
Hengjie Zheng†‡, Jie Liu*, Shuangwei Hu§, Lian Li†‡$ and Jingling Xue*$
*UNSW Sydney, Australia
†SKL of Computer Architecture, ICT, CAS, China
‡University of Chinese Academy of Sciences, China
§Vivo AI Lab, China

***Abstract*—The IFDS algorithm can be compute- and memory-intensive for some large programs, often running for a long time (more than expected) or terminating prematurely after some time and/or memory budgets have been exhausted. In the latter case, the corresponding IFDS data-flow analyses may suffer from false negatives and/or false positives. To improve this, we introduce a sparse alternative to the traditional IFDS algorithm. Instead of propagating the data-flow facts across all the program points along the program's (interprocedural) control flow graph, we propagate every data-flow fact directly to its next possible use points along its own sparse control flow graph constructed on the fly, thus reducing significantly both the time and memory requirements incurred by the traditional IFDS algorithm.**

In our evaluation, we compare FLOWDROID, a taint analysis performed by using the traditional IFDS algorithm, with our sparse incarnation, SPARSEDROID, on a set of 40 Android apps selected. For the time budget (5 hours) and memory budget (220GB) allocated per app, SPARSEDROID can run every app to completion but FLOWDROID terminates prematurely for 9 apps, resulting in an average speedup of 22.0x. This implies that when used as a market-level vetting tool, SPARSEDROID can finish analyzing these 40 apps in 2.13 hours (by issuing 228 leak warnings) while FLOWDROID manages to analyze only 30 apps in the same time period (by issuing only 147 leak warnings).

***Index Terms*—IFDS, data-flow analysis, taint analysis**

I. INTRODUCTION

The IFDS (Interprocedural, Finite, Distributive, Subset) data-flow problems formulated in [1] are solved in a wide range of application areas, including model checking [2]–[5], program verification [6]–[8], slicing [9], pointer analysis [10], dynamic test generation [11], [12], bug detection [13]–[15], security analysis [16], [17], and taint analysis [18]–[21]. In such an interprocedural data-flow problem, the set of data-flow facts D is finite and the data flow functions (in $2^D \mapsto 2^D$) distribute over the meet operator $\sqcap$ (union or intersection).

When formulating the IFDS problems, Reps et al. [1] introduced a polynomial-time algorithm for solving each as a special kind of graph-reachability problem (reachability along interprocedurally realizable paths). This IFDS algorithm operates on the interprocedural CFG $G^* = (N^*, E^*)$ of a program (consisting of the CFGs for all its functions connected by call and return edges). Interprocedurally, the data-flow facts in D are propagated from a callsite to an invoked callee `foo` in G^* (via a call edge to `foo`'s CFG) and then back from the `foo`'s CFG (via a return edge) to the same callsite, context-sensitively over a balanced-parentheses language by matching call and return edges. Intraprocedurally, the data-flow facts in D are propagated across the edges in `foo`'s CFG. This algorithm runs in $O(|E^*||D|^3)$ until a fixed point is found.

This classic IFDS algorithm can be compute- and memory-intensive, as it propagates the data-flow facts across all the program points along all the edges in G^*. Its multi-threaded versions [19], [22], [23] can speed it up, but still unsatisfactorily for some programs. For example, FLOWDROID [19] includes a multi-threaded IFDS solver that deploys multiple threads on multiple CPU cores to propagate the data-flow facts concurrently along different edges in G^*. However, when applied to perform taint analysis for a set of 2,950 Android apps on a computer server with 64 Intel Xeon CPU cores equipped with 730GB RAM, its IFDS solver (with even many compromises made) can spend 24+ hours on one app by using all the memory, resulting in 16 apps still unanalyzable [24]. Such premature terminations will cause the underlying analysis to report either more false negatives (i.e., miss more bugs) or more false positives (i.e., false warnings) or both.

In practice, static analysis tools have become part of the core developer workflow in software industries such as Google [25]. As part of nightly builds, static analysis tools are expected to be efficient, especially for handling large codebases [26].

In this paper, we focus on accelerating the IFDS algorithm [1] by propagating the data-flow facts sparsely rather than densely in G^* (orthogonally to multi-threaded acceleration). Our key insight is that the traditional IFDS algorithm propagates too many data-flow facts redundantly across too many edges in G^* before they are actually used, resulting in excessive time and memory requirements for some programs. We can speed it up significantly if we can cut down its CPU and memory usage. We will achieve this by propagating a data-flow fact *sparsely*, i.e., directly to its next possible use points in G^*. The challenge lies in how to identify such "next possible use points" for a data-flow fact efficiently, without losing the performance benefits reaped from its subsequent sparse propagation. The key novelty here is to build a sparse CFG (SCFG) for each data-flow fact on-demand (i.e., only when needed) to enable its profitable sparse propagation.

$Corresponding author

To demonstrate the performance benefits for sparsifying the IFDS algorithm, we consider taint analysis for finding information leaks in Android apps, a significant client analysis solved as an IFDS problem. Currently, FLOWDROID [19] represents a state-of-the-art tool for solving taint analysis by using a multi-threaded implementation of the traditional IFDS algorithm [1]. FLOWDROID applies its two passes iteratively, a forward one for discovering more tainted access paths and a backward one for discovering more aliases, until a fixed point has been reached. As its sparse alternative, our new tool, SPARSEDROID, aims to improve its performance by employing our sparse IFDS algorithm in both of its two passes.

This paper makes the following main contributions:

- We present a new approach for sparsifying the traditional IFDS problems to accelerate its performance.
- We introduce a sparse IFDS-based taint analysis for detecting information leaks in Android apps.
- We have evaluated SPARSEDROID against FLOWDROID (driven by a non-sparse IFDS algorithm) on 40 Android apps (with 34 from FossDroid [27] and 6 from Google Play). Given a time budget (5 hours) and memory budget (220GB) per app, SPARSEDROID runs every app to completion but FLOWDROID terminates prematurely for 9 apps, resulting in an average speedup of 22.0x. This implies that when used as a vetting tool, SPARSEDROID can finish analyzing these 40 apps in 2.13 hours by issuing 228 leak warnings while FLOWDROID analyzes only 30 apps in the same time period by issuing only 147 leak warnings (even if "timeout apps" are ignored).

The rest of the paper is organized as follows. Section II provides an overview of and motivates our sparse IFDS analysis. Section III introduces our sparse IFDS framework. Section IV contains a concrete instantiation of our sparse framework for performing sparse taint analysis. Section V presents and analyzes our experimental results. Section VI describes the limitations of our sparse algorithm. Section VII discusses the related work. Finally, Section VIII concludes.

II. MOTIVATION

We motivate our sparsification of the IFDS algorithm [1] by considering taint analysis as an instantiation. In Section II-A, we introduce an example program and find its tainted access paths manually. In Section II-B, we describe how FLOWDROID [19] does this automatically by applying the traditional IFDS algorithm. In Section II-C, we explain how SPARSEDROID, our sparse version of FLOWDROID, works.

A. An Example Program

Let us focus on analyzing `foo()` in Figure 1, by assuming that `a` and `a.g` passed from its callers are untainted and the call to `A()` (line 12) has no effect on our analysis. To find all the tainted accesses manually, we need to keep track of all the relevant access paths (data-flow facts) flowing along the control flow while also being mindful about their aliases.

Initially, `b` is tainted in line 2. Then both `a.f` (an instance field) and `A.h` (a static variable) are tainted in lines 3 and 4, respectively. However, `A.h` becomes untainted after each call to `A.bar()` (lines 7 and 10) as it is killed in `A.bar()`. In line 14, as `a.f` is tainted, `x.k.f` gets tainted. As `c` and `x` are aliases, `c.k.f` is tainted in line 14, too. With k-limiting [28], all the access paths that are longer than k are truncated.

B. FLOWDROID*: The IFDS-based Taint Analysis*

FLOWDROID [19] applies the traditional IFDS algorithm to solve the taint analysis problem, as illustrated in Figure 1(a). FlowDroid operates on the interprocedural CFG (ICFG), i.e., supergraph $G^* = (N^*, E^*)$ [1], of the program. G^* consists of the CFGs of all the functions in the program. There are four types of edges. In addition to the *normal*, i.e., intraprocedural edges in a CFG, every callsite is connected to a called function by a *call edge*, a *return edge*, and a *call-to-return edge* (to enable the interprocedural data-flows across the callsite). In actuality, G^* is exploded by associating its edges with all the data-flow facts (i.e., access paths) in D tracked during the taint analysis. The meet operator $\sqcap$ used is $\cup$ as an access path is considered as tainted at a joint point if it is tainted in any of incoming control-flow paths. Each edge has a flow function that takes the set of old facts arriving at the edge as input and sends the set of new facts across the edge as output.

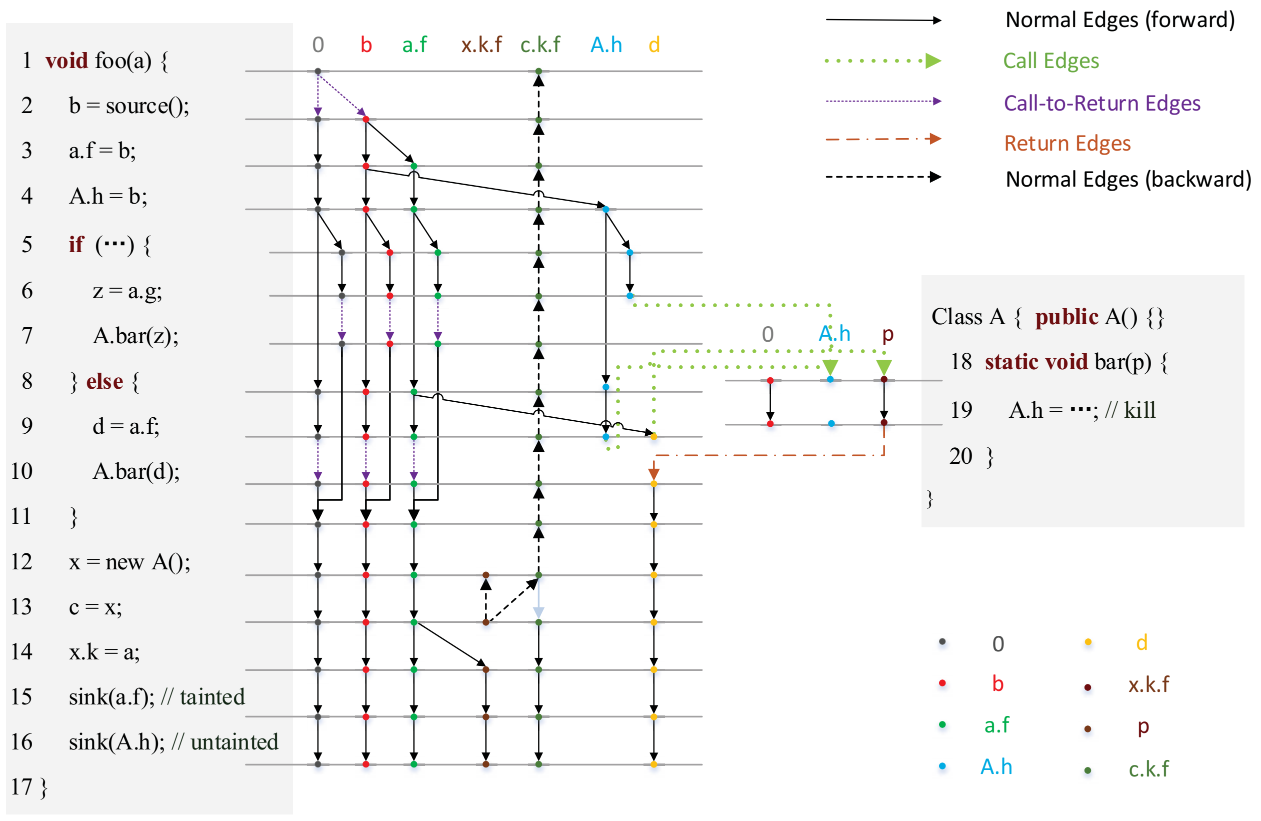
FLOWDROID proceeds iteratively by performing two passes repeatedly, a forward pass for discovering tainted accesses and a backward pass for dicovering aliases, until a fixed point is reached. In the first forward pass, FLOWDROID discovers `b`, `a.f`, `A.h`, `x.k.f` and `d` to be tainted at different program points except that `A.h` becomes untainted after each call to `A.bar()`. For example, as `a.f` is a tainted fact in line 14, the flow function associated with its edge ending at line 15 will cause `x.k.f` to be tainted. In the IFDS algorithm [1], **0** denotes an empty fact allowing new facts to be generated at a program point (e.g., line 2). As `x.k = a` is a store, FLOWDROID starts a backward pass to search for the aliases of `x.k.f`, finding `c.k.f` to be aliased with `x.k.f` after line 13. For this standard pass, we refer to [19] for more details. In a subsequent forward pass, `c.k.f` is propagated forwards and recognized as being tainted just after line 14.

Two points are made about the traditional IFDS algorithm:

- **Dense Propagation of Facts.** The traditional IFDS algorithm [1] employed by FLOWDROID propagates many data-flow facts across many program points redundantly before they are actually used, as highlighted by many such solid dots in Figure 1(a). Take `a.f`, tainted in line 3, for example. All the subsequent propagations before it reaches line 9 and 14 are a waste of both CPU and memory resources.
- **Multi-Threading.** In the IFDS framework, all the flow functions are distributive. As also in [22], [23], FLOWDROID [19] takes advantage of multi-threading to process propagations for different edges in different threads.

C. SPARSEDROID*: Our Sparse IFDS-based Taint Analysis*

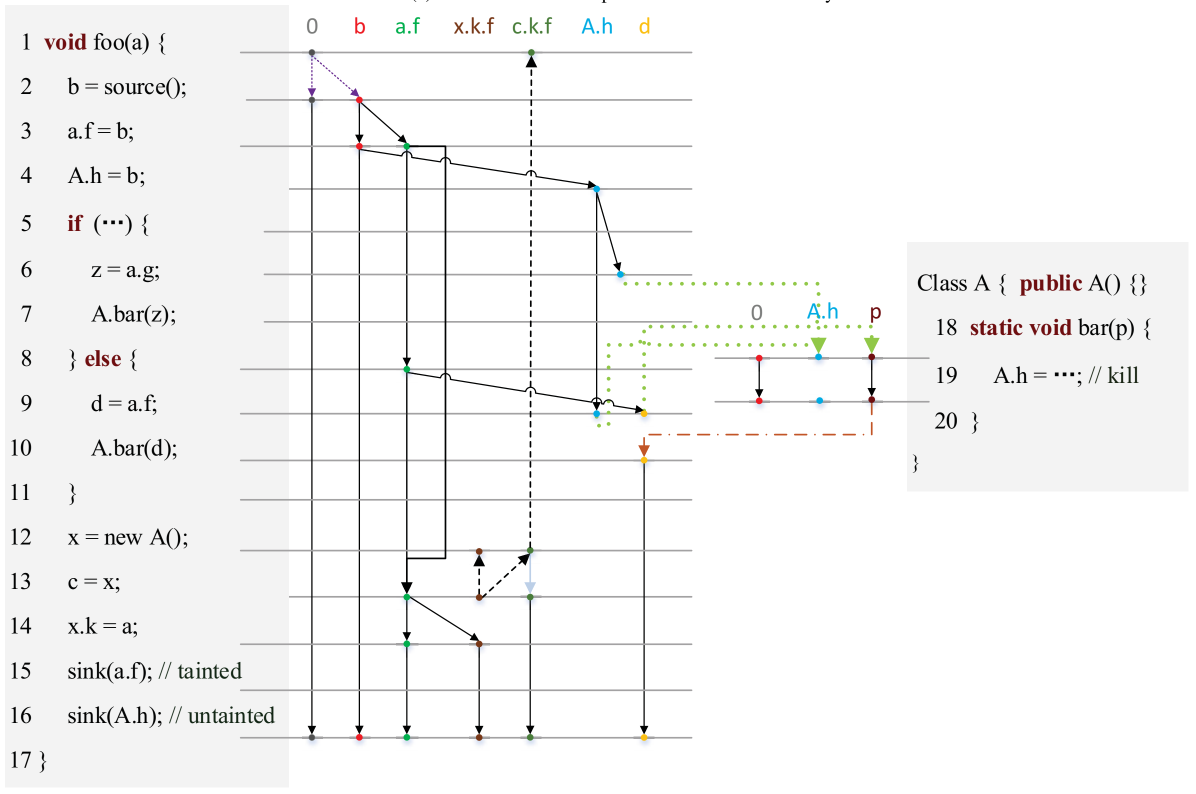
Instead of propagating data-flow facts densely as in FLOWDROID based on the traditional IFDS algorithm (Figure 1(a)), our sparse version, SPARSEDROID, based on our sparse IFDS



```
1  void foo(a) {
2    b = source();
3    a.f = b;
4    A.h = b;
5    if (···) {
6      z = a.g;
7      A.bar(z);
8    } else {
9      d = a.f;
10     A.bar(d);
11   }
12   x = new A();
13   c = x;
14   x.k = a;
15   sink(a.f); // tainted
16   sink(A.h); // untainted
17 }

Class A {  public A() {}
  18  static void bar(p) {
  19    A.h = ···; // kill
  20  }
}
```

(a) FlowDroid: non-sparse IFDS-based taint analysis

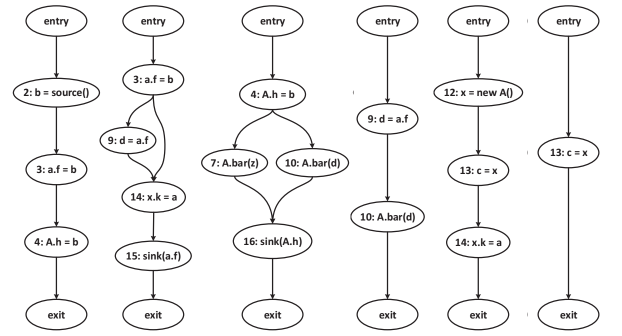


```
1  void foo(a) {
2    b = source();
3    a.f = b;
4    A.h = b;
5    if (···) {
6      z = a.g;
7      A.bar(z);
8    } else {
9      d = a.f;
10     A.bar(d);
11   }
12   x = new A();
13   c = x;
14   x.k = a;
15   sink(a.f); // tainted
16   sink(A.h); // untainted
17 }

Class A {  public A() {}
  18  static void bar(p) {
  19    A.h = ···; // kill
  20  }
}
```

(b) SparseDroid: sparse IFDS-based taint analysis

Fig. 1. Comparing non-sparse and sparse IFDS-based taint analysis, by discovering tainted access paths (aliases) in a forward (backward) pass iteratively.



(a) b (b) `a.f` (c) `A.h` (d) `d` (e) `x.k` (f) `c.k`

Fig. 2. The SCFGs built on-demand by SPARSEDROID for Figure 1(b).

algorithm (Figure 1(b)) will propagate them sparsely, only to where they are needed, reducing significantly the time and memory requirements (with much fewer dots remaining now).

Observation 1. *Even if the flow function associated with an edge is not the identity function (which will simply let all facts pass through the edge unchanged), many data-flow facts will still pass through the edge unchanged, without affecting the other facts and being affected by the other facts. In this case, we can simply propagate such facts sparsely, i.e., directly to their next possible use points (in the same function).*

When propagating a data-flow fact d in the CFG G_p of a function p, we exploit this observation by building on-demand a sparse CFG (SCFG) for d, $G_{p,d}$, as a sub-graph of G_p, and then propagate d across its edges instead. For taint analysis, both its forward and backward passes are sparsified. In addition, all the access paths, $v.f_1, \cdots, v.f_1.f_2. \cdots .f_n$ share exactly the same SCFG, $G_{p,v.f_1}$ (Theorem 2).

SPARSEDROID works sparsely as follows. Let us consider the first forward pass. When `b` is tainted in line 2, SPARSEDROID builds $G_{\texttt{foo,b}}$ in Figure 2(a), which contains all the statements where b is used (accessed). According to $G_{\texttt{foo,b}}$, `b` should be sent to lines 3 and 4 first and then directly to the exit of `foo()`. Similarly, in line 3, where `a.f` is tainted, we build $G_{\texttt{foo,a.f}}$ in Figure 2(b), by including the statements where `a` or `a.f` is used. This allows `a.f` to be propagated directly to lines 9 and 14. In line 4, we build $G_{\texttt{foo,A.h}}$ in Figure 2(c), by including the statements accessing `A.h` and all the callsites in `foo()` conservatively (since `A.h` is a global variable). In line 9, `d` is tainted, we build $G_{\texttt{foo,d}}$ in Figure 2(d).

When line 14 is analyzed, `x.k.f` is tainted. As this is a store, a backward alias analysis pass is triggered. At this point, we build $G_{\texttt{foo,x.k}}$ in Figure 2(e) in order to propagate sparsely all access paths sharing `x.k` as its prefix (Theorem 2). In line 13, we find `c.k.f` to be aliased with `x.k.f`. As `c.k.f` is new, we then build $G_{\texttt{foo,c.k}}$ in Figure 2(f) in order to propagate sparsely all access paths abstracted by `c.k`. Finally, in the last forward pass, `c.k.f` is recognized to be tainted in line 14.

A number of salient properties about our sparse IFDS framework are summarized as follows:

- **Sparsity.** While we sparsify only the intraprocedural analysis of the IFDS algorithm, our approach can be regarded as being full-sparse. Once propagated from a callsite to a callee, all the data-flow facts are propagated sparsely again.
- **On-Demand SCFG Construction.** SCFGs are built on-demand and reused later, reducing the unnecessary overheads that would be otherwise incurred in a pre-analysis.
- **Multi-Threading.** Sparsification is orthogonal to multi-threading parallelization. For example, SPARSEDROID has reduced significantly the number of edges that are concurrently processed by multiple threads in FLOWDROID.
- **Precision and Efficiency.** By sparsifying the IFDS algorithm, we maintain its precision while significantly reducing its time and memory requirements (despite the small time and space overheads incurred for managing SCFGs).

III. THE SPARSE IFDS FRAMEWORK

We describe how to sparsify the classic IFDS framework [1] in a formal setting. Section III-A reviews the classic IFDS algorithm [1]. Section III-B describes its sparsification. In Section IV, we will revisit taint analysis as an instantiation and have the opportunities to give some illustrating examples.

A. The Non-Sparse IFDS Framework

In Figure 3, we reproduce the classic IFDS algorithm from [1] (with a few notational changes). There are three cases, with the first two cases (lines 13 and 18) responsible for the interprocedural analysis and the last case (line 25) for the intraprocedural analysis. As discussed in Section II, only the intraprocedural analysis needs to be sparsified. However, for completeness, we introduce briefly the entire analysis below.

An instance IP of an IFDS problem is a five-tuple, $IP = (G^*, D, F, M, \sqcap)$, where $G^* = (N^*, E^*)$ is the *supergraph* of the program, D is a finite set of data-flow facts, $F \subseteq 2^D \to 2^D$ is a set of distributive functions, $M : E^* \to F$ is a map from G^*'s edges to data-flow functions (representing typically traditional transfer functions defined in terms of GEN and KILL [29]), and the meet operator $\sqcap$ is either union or intersection (depending on the problem modeled).

G^*, known traditionally as the interprocedural CFG (ICFG) of the program, consists of a collection of CFGs, $G_1, G_2, \cdots$ (one per function), one of which, $G_{\texttt{main}}$, represents the program's `main()`. For a function p, its CFG G_p consists of a unique *start* node s_p, a unique *exit* node e_p, and the remaining nodes representing its statements and predicates in the usual manner. However, a callsite is split into two nodes, a *call* node and a *return-site* node. G^* has four types of edges. The ordinary intraprocedural edges in an individual CFG G_p are called *normal edges*. For each callsite, with its call-node c and return-site node r, where p is called, three edges are included to capture its interprocedural data-flows: an intraprocedural *call-to-return* edge from c to r, an interprocedural *call edge* from c to s_p, and an interprocedural *return edge* from e_p to r.

To solve IP context-sensitively as a graph-reachability problem, G^* is extended into an *exploded supergraph*, $G^{\#}_{IP} = (N^{\#}, E^{\#})$, where $N^{\#} = N^* \times (D \cup \{\mathbf{0}\})$ and $E^{\#} =$

```
Algorithm Tabulate($G^{\#}_{IP}$)
1     Let $G^{\#}_{IP} = (N^{\#}, E^{\#})$
2     $PathEdge \leftarrow \{\langle s_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle s_{\mathtt{main}}, \mathbf{0}\rangle\}$
3     $WorkList \leftarrow \{\langle s_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle s_{\mathtt{main}}, \mathbf{0}\rangle\}$
4     $\mathcal{S} \leftarrow \emptyset$
5     ForwardTabulateSLRPs()
6     for $n \in N^*$ do
7         $X_n \leftarrow \{d_2 \in D \mid \exists\, d_1 \in (D \cup \{0\})$
                  s.t. $\langle s_{fun(n)}, d_1\rangle \rightarrow \langle n, d_2\rangle \in PathEdge\}$
Function Prop($e$)
8     if $e \notin PathEdge$ then
9         Insert $e$ into $PathEdge$; Insert $e$ into $WorkList$
Function ForwardTabulateSLRPs()
10    while $WorkList \neq \emptyset$ do
11        Pop $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ from $WorkList$
12        switch $n$
13            case $n \in callsites(p)$
14                for $d_3$ s.t. $\langle n, d_2\rangle \rightarrow \langle s_{callee(n)}, d_3\rangle \in E^{\#}$ do
15                    Prop($\langle s_{callee(n)}, d_3\rangle \rightarrow \langle s_{callee(n)}, d_3\rangle$)
16                for $d_3$ s.t. $\langle n, d_2\rangle \rightarrow \langle retSite(n), d_3\rangle \in (E^{\#} \cup \mathcal{S})$ do
17                    Prop($\langle s_p, d_1\rangle \rightarrow \langle retSite(n), d_3\rangle$)
18            case $n = e_p$
19                for $c \in callers(p)$ do
20                    for $d_4, d_5$ s.t. $\langle c, d_4\rangle \rightarrow \langle s_p, d_1\rangle \in E^{\#}$ and
                        $\langle e_p, d_2\rangle \rightarrow \langle retSite(c), d_5\rangle \in E^{\#}$ do
21                        if $\langle c, d_4\rangle \rightarrow \langle retSite(c), d_5\rangle \notin \mathcal{S}$ then
22                            Insert $\langle c, d_4\rangle \rightarrow \langle retSite(c), d_5\rangle$ into $\mathcal{S}$
23                            for $d_3$ s.t. $\langle s_{fun(c)}, d_3\rangle \rightarrow \langle c, d_4\rangle \in PathEdge$ do
24                                Prop($\langle s_{fun(c)}, d_3\rangle \rightarrow \langle retSite(c), d_5\rangle$)
25            case $n \in (N_p - callsites(p) - \{e_p\})$
26                for $\langle m, d_3\rangle$ s.t. $\langle n, d_2\rangle \rightarrow \langle m, d_3\rangle \in E^{\#}$ do
27                    Prop($\langle s_p, d_1\rangle \rightarrow \langle m, d_3\rangle$)
```

Fig. 3. The IFDS algorithm from [1] (with some notational changes).

$\{\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle \mid (m, n) \in E^*, d_2 \in M(m, n)(d_1)\}$. Note that **0** signifies an empty set of facts (allowing new facts to be generated at a program point) and $M(m, n) \in F$ is the flow function associated with the edge $(m, n) \in E^*$. For efficiency reasons, $G^{\#}_{IP}$ is usually built from G^* at run time.

In Figure 3, an IP is solved with dynamic programming in $O(|E^*||D|^3)$. For a function p, $callsites(p)$ is the set of its call statements and $callers(p)$ is the set of call statements (in p's callers) where p is invoked. For a call statement n, $retSite(n)$ denotes its return-side node and $callee(n)$ the function invoked. For a node $n \in N^*$, $fun(n)$ identifies its containing function. Finally, $\mathcal{S}$ summarizes the interprocedural data-flow facts obtained across the function boundaries.

$PathEdge$ records the set of *path edges*, with each $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ representing the suffix of a realizable path from $\langle s_{\mathtt{main}}, \mathbf{0}\rangle$ to $\langle n, d_2\rangle$, implying that the path edge from $\langle s_{\mathtt{main}}, \mathbf{0}\rangle$ to $\langle s_p, d_1\rangle$ is realizable. Starting with $\langle s_{\mathtt{main}}, \mathbf{0}\rangle \rightarrow \langle s_{\mathtt{main}}, \mathbf{0}\rangle$ (lines 2 – 4), `ForwardTabulateSLRPs()` collects all possible path edges in $PathEdge$ (line 5) with a case analysis, by handling (1) the interprocedural data-flows entering a function (lines 13 – 17), (2) the interprocedural data-flows leaving a function (lines 18 – 24), and (3) the intraprocedural data-flows within a function (lines 25 – 27). Finally, a data-flow fact $d \in D$ exists at a program point

```
Function ForwardTabulateSLRPs()
1     while $WorkList \neq \emptyset$ do
          ...
2         switch $n$ do
13            case $n \in callsites(p)$ do
                  ...
16                for $d_3$ s.t. $\langle n, d_2\rangle \rightarrow \langle retSite(n), d_3\rangle \in (E^{\#} \cup \mathcal{S})$ do
17a                   if $\mathcal{G}^{\#}_{p,d_3}$ is not constructed yet, i.e., not in the SCFG cache then
17b                       Build $\mathcal{G}^{\#}_{p,d_3} = (\mathcal{N}^{\#}_{p,d}, \mathcal{E}^{\#}_{p,d})$ according to (2)
17c                   for $\langle n, d_2\rangle \rightarrow \langle m', d_3\rangle \in \mathcal{E}^{\#}_{p,d_3}$ do
17d                       Prop($\langle s_p, d_1\rangle \rightarrow \langle m', d_3\rangle$)
18            case $n = e_p$ do
                  ...
25            case $n \in (N_p - callsites(p) - \{e_p\})$ do
26                for $\langle \_, d_3\rangle$ s.t. $\langle n, d_2\rangle \rightarrow \langle \_, d_3\rangle \in E^{\#}$ do
27a                   if $\mathcal{G}^{\#}_{p,d_3}$ is not constructed yet, i.e., not in the SCFG cache then
27b                       Build $\mathcal{G}^{\#}_{p,d_3} = (\mathcal{N}^{\#}_{p,d}, \mathcal{E}^{\#}_{p,d})$ according to (2)
27c                   for $\langle n, d_2\rangle \rightarrow \langle m', d_3\rangle \in \mathcal{E}^{\#}_{p,d_3}$ do
27d                       Prop($\langle s_p, d_1\rangle \rightarrow \langle m', d_3\rangle$)
```

Fig. 4. The sparse IFDS algorithm adapted from Figure 3 with its line 17 (line 27) replaced by lines 17a – 17d (lines 27a – 27d).

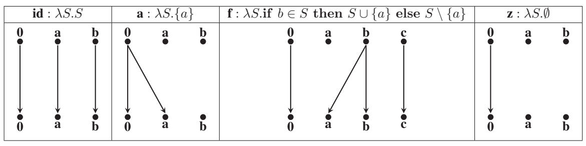

id : $\lambda S.S$
a : $\lambda S.\{a\}$
f : $\lambda S.$**if** $b \in S$ **then** $S \cup \{a\}$ **else** $S \setminus \{a\}$
z : $\lambda S.\emptyset$
0
a
b
c

Fig. 5. Edge-specific flow functions illustrated (with the first two from [1]).

$n \in N^*$ iff there exists a realizable path in $G^{\#}_{IP}$ from node $\langle s_{\mathtt{main}}, \mathbf{0}\rangle$ to node $\langle n, d\rangle$ (lines 6 – 7).

For our taint analysis problem, $IP = (G^*, D, F, M, \sqcap)$, Figure 1(a) gives its exploded supergraph. As explained in Section II, D is the set of access paths, $\sqcap$ is $\cup$, and F and M were discussed earlier but will be formalized in Section IV.

B. Sparsification

As the IFDS algorithm in Figure 3 spends the majority of its analysis time on propagating data-flow facts intraprocedurally, it suffices to sparsify its intraprocedural analysis only. Our sparse algorithm, which is given in Figure 4, is conceptually simple and easy to implement as well as both time- and space-efficient. Let us describe our sparsification below.

Consider any IFDS problem $IP = (G^*, D, F, M, \sqcap)$. According to Observation 1, *even if the flow function $M(m, n) \in 2^D \mapsto 2^D$ is not the identity function, many data-flow facts $X \subseteq D$ are propagated through (m, n) unchanged, without affecting the other facts and being affected by the other facts. Our key insight is to propagate such facts in X sparsely, i.e., directly to their next possible use points (in the same function), thereby reducing both time and space requirements overall.*

Figure 5 illustrates a few edge-specific flow functions, which can be understood as traditional transfer functions composed using GEN and KILL [29]. Here, **id** is the identity function, **a** generates a new fact a but kills everything else in S, **f** generates (kills) a if $b \in S$ ($b \notin S$), and **z** kills everything in S. If $M(m, n) = $ **id**, we can avoid propagating all the facts

]

along (m, n) by sending them directly to their next use points. However, even if $M(m, n) \neq \mathbf{id}$, such sparse propagations are still possible for some facts (as motivated in Figure 1). To achieve this, we introduce fact-specific identity functions.

Let $f \in F \subseteq 2^D \mapsto 2^D$ be a flow function associated with edge (m, n). Let $d \in D$ be a fact. We say that f is a *d-specific identity function*, denoted $f \equiv f^d$, if the following holds:

$$\begin{array}{rcl} \forall\ X \in 2^D & : & d \in X \Longrightarrow d \in f(X) \\ \forall\ X \in 2^{D \setminus \{d\}} & : & f(X) \setminus \{d\} = f(X \cup \{d\}) \setminus \{d\} \end{array} \tag{1}$$

For any fact d arriving at m, d will emerge at n without being affected by the other facts by the first condition, and d does not affect the other facts reaching n by the second condition.

Let us re-examine each flow function (associated with an edge $(m, n) \in E^*$) in Figure 5. As $\mathbf{id}$ is an identity function by itself, $\mathbf{id}$ is a d-specific identity function for any d. For $\mathbf{a}$, we have $\mathbf{a} \equiv \mathbf{a}^a$ (as a that reaches m is regenerated and propagated to n) but $\mathbf{a} \not\equiv \mathbf{a}^b$ (as b is killed). For $\mathbf{f}$, it is not hard to see that $\mathbf{f} \not\equiv \mathbf{f}^a$, $\mathbf{f} \not\equiv \mathbf{f}^b$ but $\mathbf{f} \equiv \mathbf{f}^c$. Finally, $\mathbf{z}$ is not a d-specific identity function for any d (as all facts are killed).

For a call statement n, there is a unique intraprocedural edge from n to its return-site node, $retSite(n)$. Whether $M(n, retSite(n))$ is a d-specific identity function, where $d \in D$, depends on the underlying IFDS problem, as will be discussed in Section IV. Conservatively, we can always assume $M(n, retSite(n)) \not\equiv M(n, retSite(n))^d$ for every $d \in D$.

Given $d \in D$, we are motivated to propagate d sparsely across a CFG. Let $G_p = (N_p, E_p)$ be the CFG of a function p in G^*. Let $G_p^\# = (N_p^\#, E_p^\#)$ be the exploded CFG of G_p in G_{IP}. We will propagate d in a sparse CFG (SCFG), $\mathcal{G}_{p,d} = (\mathcal{N}_{p,d}, \mathcal{E}_{p,d})$, as a subgraph of G_p. To construct $\mathcal{G}_{p,d}$, we construct below its exploded version $\mathcal{G}_{p,d}^\# = (\mathcal{N}_{p,d}^\#, \mathcal{E}_{p,d}^\#)$ (as a subgraph of $G_p^\#$). In practice, $\mathcal{G}_{p,d}$ is actually built first and then exploded into $\mathcal{G}_{p,d}^\#$. For convenience, we assume the existence of a pseudo start (exit) node s'_p (e'_p) in $G_p^\#$, such that $M(s'_p, s_p) \not\equiv M(s'_p, s_p)^d$ and $M(e_p, e'_p) \not\equiv M(e_p, e'_p)^d$.

Let $P_p^d(n_1, n_k) =_{\text{def}} (n_0, n_1), \cdots, (n_k, n_{k+1})$ be a *sparsifiable path* in $G_p^\#$, where $k \geqslant 2$, such that (1) $M(n_0, n_1) \not\equiv M(n_0, n_1)^d$, (2) $M(n_i, n_{i+1}) \equiv M(n_i, n_{i+1})^d$, where $1 \leqslant i < k$, and (3) $M(n_k, n_{k+1}) \not\equiv M(n_k, n_{k+1})^d$. Starting at n_1, we will send d directly to n_k by skipping these intermediate edges.

The SCFG $\mathcal{G}_p^d = (\mathcal{N}_{p,d}^\#, \mathcal{E}_{p,d}^\#)$ is defined as follows:

$$\begin{array}{rl} \mathcal{E}_{p,d}^\# = & \{(m, d) \to (n, d') \in E_p^\#) \mid M(m, n) \not\equiv M(m, n)^d\} \\ & \cup\ \{(m, d) \to (n, d) \mid P_p^d(m, n) \text{ is sparsifiable}\} \\ \mathcal{N}_{p,d}^\# = & \bigcup_{(m,d) \to (n,d') \in \mathcal{E}_{p,d}^\#} \{(m, d)\} \cup \{(n, d')\} \end{array} \tag{2}$$

For our motivating example, we examined a few SCFGs in Figure 2 and will discuss them in more detail in Section IV.

Frequently, many facts share the same SCFG (as motivated in Section II and discussed further in Section IV). If $G_{p,d_1} = G_{p,d_2}$, then the same SCFG is shared by d_1 and d_2.

We can now understand our sparse algorithm given in Figure 4 easily. For reasons of symmetry, it suffices to explain its lines 27a – 27d. Whenever the non-sparse IFDS algorithm is just about to propagate a fact d_3 at node m densely (line 27 of Figure 3), our sparse algorithm propagates d_3 on its own SCFG $\mathcal{G}_{p,d_3}^\#$ (built on the fly but reused subsequently) sparsely.

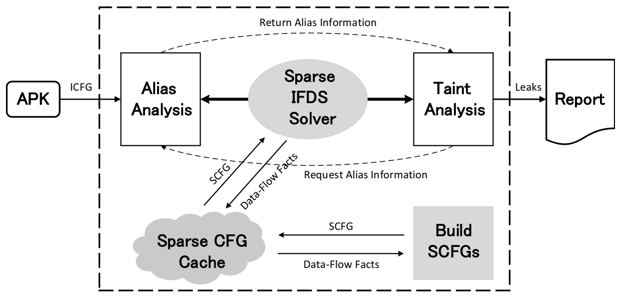


Fig. 6. The workflow of SPARSEDROID.

Theorem 1 (Correctness)**.** *The sparse algorithm (Figure 4) computes the same results as the non-sparse one (Figure 3).*

Proof Sketch. Follows from the fact that every data-flow fact d_3 is now propagated sparsely in $\mathcal{G}_{p,d_3}^\#$ (Figure 4) rather than densely in $G_p^\#$ (Figure 3) by skipping only the statements that are irrelevant as far as d_3 is concerned by (1) – (2). □

IV. THE SPARSE TAINT ANALYSIS

We describe an instantiation of our sparse algorithm for finding tainted accesses in Android apps. Figure 6 depicts SPARSEDROID, our sparse version of FLOWDROID [19]. We reuse FLOWDROID's modules, "Taint Analysis" and "Alias Analysis", for discovering tainted accesses and aliases in the forward and backward passes, respectively (Section II). The three modules with the light gray background are added by us to support sparse analysis. The "Sparse IFDS Solver" implements our sparse IFDS algorithm (Figure 4), which is multi-threaded exactly as in FLOWDROID. The "Sparse CFG Cache" manages all the SCFGs constructed so far while the "Build SCFGs" module constructs an SCFG for a data-flow fact on the fly if it not cached yet according to (2).

For the IFDS-based taint analysis, let us revisit its five-tuple: $IP = (G^*, D, F, M, \sqcap)$. In Figure 6, G^* is the ICFG, i.e., supergraph provided to the analysis. Note that D is the set of access paths and $\sqcap$ is $\cup$. In Section II, we discussed informally the flow functions in F that are mapped to the edges in G^* by M. In this section, we will define what M is precisely.

Section IV-A introduces the intermediate representation (IR) used. Section IV-B formalizes the flow functions used for taint analysis. Section IV-C discusses SCFG construction.

A. Intermediate Representation

Table I gives eight kinds of statements used in our IR. By convention, $\overline{x} =_{\text{def}} x_0, \cdots, x_{n-1}$ is a possibly empty list of elements. To handle both virtual and static calls uniformly, a virtual call $a_0.foo(a_1, \cdots, a_{n-1})$ is written as $foo(a_0, \cdots, a_{n-1})$ and a static call $T.foo(a_0, \cdots, a_{n-1})$ is written as $foo(a_0, \ldots, a_{n-1})$ by dropping the irrelevant T.

All local variables are assumed to be in SSA form [30], resulting in the Phi instruction being used as is standard.

$$\frac{a = \mathit{source}() \quad \{\mathbf{0}\}}{\{\mathbf{0}, a.*\}} \quad \frac{a = \mathit{source}() \quad \{v.\bar{f}\} \quad v \neq a}{\{v.\bar{f}\}} \quad \text{[SOURCE]} \qquad \frac{a = \ldots \quad \{v.\bar{f}\} \quad v = a}{\{\}} \quad \text{[KILL]}$$

$$\frac{a = \mathit{new}\ T() \quad \{v.\bar{f}\} \quad v \neq a}{\{v.\bar{f}\}} \quad \text{[NEW]} \qquad \frac{a = b \quad \{v.\bar{f}\} \quad v = b}{\{v.\bar{f}, a.\bar{f}\}} \quad \frac{a = b \quad \{v.\bar{f}\} \quad v \notin \{a, b\}}{\{v.\bar{f}\}} \quad \text{[ASSIGN]}$$

$$\frac{a_2 = \phi(a_0, a_1) \quad \{v.\bar{f}\} \quad v \in \{a_0, a_1\}}{\{v.\bar{f}, a_2.\bar{f}\}} \quad \frac{a_2 = \phi(a_0, a_1) \quad \{v.\bar{f}\} \quad v \notin \{a_0, a_1, a_2\}}{\{v.\bar{f}\}} \quad \text{[PHI]}$$

$$\frac{a = \xi.f' \ (\xi \in \{b, T\}) \quad \{v.\bar{f}\} \quad v = \xi \wedge \mathbf{car}(\bar{f}) = f'}{\{v.\bar{f}, a.\mathbf{cdr}(\bar{f})\}} \quad \frac{a = \xi.f' \ (\xi \in \{b, T\}) \quad \{v.\bar{f}\} \quad v \neq a \wedge (v \neq \xi \vee \mathbf{car}(\bar{f}) \neq f')}{\{v.\bar{f}\}} \quad \text{[LOAD]}$$

$$\frac{\begin{array}{c}\xi.f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \\ v = b\end{array}}{\{v.\bar{f}, \xi.f'.\bar{f}\}} \quad \frac{\begin{array}{c}\xi.f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \\ v = \xi \wedge \mathbf{car}(\bar{f}) = f'\end{array}}{\{\}} \quad \frac{\xi.f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \quad v \neq b \wedge (v \neq \xi \vee \mathbf{car}(\bar{f}) \neq f')}{\{v.\bar{f}\}} \quad \text{[STORE]}$$

$$\frac{\begin{array}{c} r = \mathit{foo}(\bar{a}) \quad \{v.\bar{f}\} \\ v = a_i \quad a_i \in \bar{a} \quad p_i \text{ is } a_i\text{'s corresponding formal parameter in } \mathit{foo}\end{array}}{\{p_i.\bar{f}\}} \quad \frac{r = \mathit{foo}(\bar{a}) \quad \{v.\bar{f}\} \quad v \notin \bar{a}}{\{\}} \quad \frac{r = \mathit{foo}(\bar{a}) \quad \{T.\bar{f}\}}{\{T.\bar{f}\}} \quad \text{[CALL]}$$

$$\frac{r = \mathit{foo}(\bar{a}) \quad \{v.\bar{f}\} \quad v \in \bar{a} \vee v = T}{\{\}} \quad \frac{r = \mathit{foo}(\bar{a}) \quad \{v.\bar{f}\} \quad v \notin \bar{a} \cup \{r\} \wedge v \neq T}{\{v.\bar{f}\}} \quad \text{[CALL-TO-RETURN]}$$

$$\frac{\begin{array}{c} \mathit{ret}_{\mathit{foo}}\ r \quad \{v.\bar{f}\} \quad v = p_i \\ p_i \text{ is } \mathit{foo}\text{'s formal parameter} \quad a_i \text{ is } p_i\text{'s corresponding actual argument}\end{array}}{\{a_i.\bar{f}\}} \quad \frac{\begin{array}{c} \mathit{ret}_{\mathit{foo}}\ r_0 \quad \{v.\bar{f}\} \quad v = r_0 \\ r_1 = \mathit{foo}(\bar{a}) \in \mathit{callers}(\mathit{foo})\end{array}}{\{r_1.\bar{f}\}} \quad \frac{\mathit{ret}_{\mathit{foo}}\ r \quad \{T.\bar{f}\}}{\{T.\bar{f}\}} \quad \text{[RETURN]}$$

Fig. 7. The flow functions for taint analysis operating on the IR given in Table I (with the alias analysis performed orthogonally [19] as illustrated in Figure 1).

TABLE I
INTERMEDIATE REPRESENTATION FOR TAINT ANALYSIS.

Name	**Instruction**	**Name**	**Instruction**
Source	$a = \mathit{source}()$	Phi	$a_2 = \phi(a_0, a_1)$
New	$a = \mathit{new}\ T()$	Assign	$a = b$
Load	$a = b.f$	Store	$a.f = b$
	$a = T.f$		$T.f = b$
Call	$r = \mathit{foo}(\bar{a})$	Return	return$_p$ r

B. Flow Functions

Figure 7 gives all the flow functions used for taint analysis in our IR. In the IFDS framework, a flow function $\mathcal{F} \in 2^D \mapsto 2^D$, where D is the set of access paths, distributes over the meet operator $\sqcap$: $\mathcal{F}(X) = \sqcap_{x \in X} \mathcal{F}(\{x\})$. Given a flow function $\mathcal{F}$ associated with a control-flow edge from a statement s at a source node to a statement s' at a target node in the CFG, a rule for s maps the singleton fact set $\{x\}$ on entry of s to $\mathcal{F}(\{x\})$ on entry of s'. (If x is tainted, then $\mathcal{F}(\{x\})$ contains all the tainted facts by s.) For a non-source statement, $\mathcal{F}(\mathbf{0}) = \mathbf{0}$ is omitted. For a list $\bar{x}$, $\mathbf{car}(\bar{x})$ and $\mathbf{cdr}(\bar{x})$ return the first and the rest of the list, respectively. To simplify the presentation of our rules, $v.\bar{f}$ denotes an access path (i.e., a local variable) v or a field access $v.f_1.\cdots.f_n$, where $n \geqslant 1$ and $T.\bar{f}$ denotes an access path $T.f_1.\cdots.f_n$, where $n \geqslant 1$.

The rules for handling non-call statements are simple. At a tainted source, a = source(), $a.*$ signifies that a and all its associated access paths are tainted. In [LOAD] and [STORE], if $v.\bar{f} = v.*$, then $\mathbf{car}(*) = f'$ is assumed to hold always and $a.\mathbf{cdr}(*) = a.*$. Now, consider the three rules for handling a call statement, [CALL], [CALL-TO-RETURN] and [RETURN]. There are two cases in handling $r = \mathit{foo}(\bar{a})$. A fact that is a static variable, `T.f`, is propagated to its entry and then its exit (if it remains tainted at the end). A fact that is an instance variable, `v.f`, is handled in the same way if `v` is one of `foo`'s arguments and propagated to the call's return node otherwise.

C. SCFG Construction

As motivated in Section II, we will construct an SCFG $G_{p,d}$ on-demand for every new fact d generated when a function p is analyzed. Our sparse IFDS algorithm given in Figure 4 will then operate on $G^{\#}_{p,d}$, which is exploded from $G_{p,d}$ according to (2) during the sparse analysis (Section III-B).

For a given IFDS problem, many facts may have identical SCFGs. For taint analysis, we rely on the following theorem.

Theorem 2. *Let $\xi.f_1.\cdots.f_n$ be an access path, where $\xi \in \{a, T\}$. Then $G_{p,\xi.f_1} = G_{p,\xi.f_1,\cdots,f_i}$, where p is any function.*

Proof. Follows from Figure 7 (as all the field accesses in the IR are limited to only a single field as shown in Table I). $\square$

Therefore, we only need to build SCFGs for three types of data-flow facts: (1) a local variable v, (2) a field access $v.f$, and (3) a global variable $T.f$. Let $G_p = (N_p, E_p)$ be the CFG of a function p. To build these SCFGs (according to (2)), it is just a simple matter of going through the flow functions $M(m,n)$ for all the edges $(m,n) \in E$ and checking to see if each is a fact-specific identity function or not (by (1)):

- $G_{p,v}$. $M(m,n) \not\equiv M(m,n)^v$ iff v appears in the statement at node m. In this case, $G_{p,v}$ consists of all the nodes in G_p, where v is referred to (Figures 2(a) and (d)).
- $G_{p,v.f}$. $M(m,n) \not\equiv M(m,n)^{v.f}$ iff the statement at m refers to v alone (without a field access) or $v.f$. Thus, $G_{p,v.f}$ consists of all such nodes in G_p (Figures 2(b), (e) and (f)).
- $G_{p,T.f}$. $M(m,n) \not\equiv M(m,n)^{T.f}$ iff the statement at m accesses $T.f$ or is a callsite (as $T.f$ may be killed indirectly). Thus, $G_{p,T.f}$ consists of all such nodes in G_p (Figure 2(c)).

Finally, there is one FLOWDROID-specific implementation detail concerning when to "activate" aliases as being tainted. Consider Figure 1 again. During a forward analysis, `x.k.f` is found to be tainted by `x.k = a` in line 14. Then a backward pass is started to search for its aliases. In line 13, `c.k.f` is found as an alias. In a subsequent forward pass, `c.k.f` is propagated forward and recognized as being tainted after it has passed `x.k = a`, its so-called *activation statement*.

In SPARSEDROID, we propagate `c.k.f` in $G_{\texttt{foo,c.k}}$ shown in Figure 2 rather than $G_{\texttt{foo}}$. If its activation statement `x.k = a` does not appear in $G_{\texttt{foo,c.k}}$, which is true in this case, we can simply activate `c.k.f` in the first nodes in $G_{\texttt{foo,c.k}}$ that are reachable from this activation statement in $G_{\texttt{foo}}$ (along its outgoing paths). The exit in $G_{\texttt{foo,c.k}}$ (and $G_{\texttt{foo}}$) satisfies this condition. Note that by construction, `c.k.f` will simply be propagated through lines 15 and 16 before reaching the exit.

V. EVALUATION

We demonstrate the significant performance benefits achieved by our sparse IFDS algorithm by comparing SPARSEDROID and FLOWDROID for solving taint analysis in Android apps as a major application. Our sparse analysis is fairly general. Other possible applications include pointer analysis [10], typestate analysis [31], uninitialized variables [1], constant propagation [32], and Android compatibility detection [33].

For the taint analysis problem as motivated in Figure 1, FLOWDROID [19] relies on the traditional non-sparse IFDS algorithm (Figure 3) while SPARSEDROID takes advantage of our sparse IFDS algorithm (Figure 4). By Theorem 1, SPARSEDROID is equivalent to FLOWDROID in terms of their precision (i.e., leak-finding capability) except that SPARSEDROID is sparse. We have validated the correctness of SPARSEDROID by extensive benchmarking. In the case of DROIDBENCH [34] (containing small unit tests with each consisting of dozens of lines of code), for example, SPARSEDROID has successfully passed all the test cases that FLOWDROID has passed (with a caveat on implicit data flows discussed in Section VI). We will therefore focus on evaluating the performance advantages of SPARSEDROID over FLOWDROID, by using relatively large Android benchmarks and real-world Android apps.

Our evaluation, as described below, will attempt to address the following four research questions:

- **RQ1.** Is SPARSEDROID faster?
- **RQ2.** Is SPARSEDROID more memory-efficient?
- **RQ3.** Is the sparse IFDS algorithm effective?
- **RQ4.** Is the on-demand SCFG construction effective?

A. Experimental Setup

a) **Implementation:** For FLOWDROID, we use its latest version (`0967ca2`) [35] implemented in Soot [36]. SPARSEDROID is a sparse version (Figure 6). Given an app, its Dalvik bytecode is converted into Soot's Shimple IR in SSA form, operated on by both tools. In FLOWDROID, all access paths are abstracted with k-limiting with $k = 5$ (by default). We also use its source and sink definitions for taint analysis.

b) **Benchmarks:** We have selected 40 Android apps, with 34 from Fossdroid [27], an alternate web interface for the F-Droid repository [37], and 6 from Google Play. From Fossdroid, we pick randomly 2 apps from the top 20 most popular apps in each of its 17 categories. Note that `Graphics` has 17 Apps in total. From `Google Play`, we obtain `pokemongo` (a game app), `word` (a Microsoft Word app), `outlook` (a Microsoft Outlook app), `reader` (an Adobe Acrobat reader), `bk` (a Burger King app), and `oeffi` (a journey planner).

c) **Platform:** Our experiments are done on an Intel Xeon E5-1660 v4 CPUs (3.20GHz) server with 256GB RAM, running Ubuntu 16.04.4 LTS (Xenial Xerus). For JVM, we set the maximum heap size as 220GB (with `-Xmx`). For both tools, we turn on `--mergedexfiles` to enable analyzing multiple dex files for an app and use the `-dt` option to set 5 hours as the per-app time budget for their IFDS solvers. As our platform has 8 cores, 8 threads are used to process all the path edges in $WorkList$ in Figures 3 and 4.

d) **Results:** Table II contains the results for FLOWDROID (FD) and SPARSEDROID (SD). The analysis time of an app is the average of 3 runs. As indicated in its caption, FLOWDROID terminates prematurely in 9 apps (with 3 running out of time and 6 running out of memory). However, SPARSEDROID has successfully run every app to completion within the time and memory budgets given. For each app (Columns 1 – 4), we compare their analysis times (Columns 5 – 7) for RQ1, memory usage (Columns 8 – 10) for RQ2, and #PathEdges solved (Columns 11 – 13) for RQ3. Finally, we examine SCFG construction (Columns 14 – 17) for RQ4.

B. RQ1: Speedups

As shown in Columns 5 – 7 of Table II, SPARSEDROID is significantly faster than FLOWDROID, with the speedups ranging from 1.1x to 357.3x averaged to 22.0x. So SPARSEDROID outperforms FLOWDROID in every single app evaluated.

The largest speedup (357.3x) occurs on `nya.miku.wishmaster`, for which FLOWDROID spends 15,561.2 seconds to analyze it while SPARSEDROID spends only 43.6 seconds. For `org.gateshipone.odyssey` exhibiting the second largest speedup (94.0x), FLOWDROID terminates prematurely

TABLE II
PERFORMANCE OF FLOWDROID (FD) AND SPARSEDROID (SD). FLOWDROID RUNS OUT OF TIME IN THREE APPS WITH THEIR ANALYSIS TIMES INDICATED IN BLUE IN COLUMN 5 AND OUT OF MEMORY IN SIX APPS WITH THEIR MEMORY USAGE MARKED WITH OOM IN COLUMN 8.

Category	App	Version	Apk(MB)	Analysis Time (s)			Memory Usage (GB)			#PathEdges			SCFG Construction			
				FD	SD	FD/SD	FD	SD	FD/SD	FD	SD	FD/SD	#SCFGs	#Acc Paths	Time/SD (%)	Time/FD (%)
Connectivity	be.mygod.vpnhotspot	2.3.0	2.2	1,597.2	213.8	7.5	59.3	28.6	2.1	181,752,020	5,912,330	30.7	15,877	100,434	0.58	0.08
	ca.cmetcalfe.locationshare	1.3.2	0.9	14,495.5	318.7	45.5	OOM	33.1	-	826,787,619	8,621,141	95.9	17,454	170,923	0.48	0.01
Development	org.csploit.android	1.6.5	3.5	8.4	5.5	1.5	9.6	8.4	1.1	422,510	75,183	5.6	2,140	3,686	6.43	4.18
	de.k3b.android.contentproviderhelper	1.3.1.1	0.6	0.3	0.1	2.2	0.3	**0.4**	0.7	9,715	1,056	9.2	107	117	29.29	**13.06**
Games	com.ghstudios.android.mhgendatabase	2.3.1	8.8	0.2	0.1	1.3	0.5	0.4	1.2	2,148	492	4.4	154	176	38.26	**28.95**
	net.ddns.mlsoftlaberge.trycorder	5.2.3	8.6	0.2	0.1	1.4	0.4	0.3	1.3	3,491	893	3.9	147	184	17.36	**12.73**
Graphics	rodrigodavy.com.github.pixelartist	3.1	1.8	2,066.6	97.9	21.1	71.7	29.7	2.4	190,149,877	3,105,457	61.2	13,678	101,537	1.27	0.06
	uk.co.richyhbm.monochromatic	0.8.1	1.6	731.8	30.9	23.7	33.6	14.7	2.3	93,158,144	1,380,304	67.5	9,167	51,312	2.90	0.12
Internet	nya.miku.wishmaster	1.5.0	3.5	15,561.2	43.6	357.3	190.9	13.7	13.9	893,798,548	2,318,824	385.5	8,917	30,905	4.43	0.01
	acr.browser.lightning	4.5.1	2.6	77.6	25.4	3.1	38.5	15.8	2.4	12,819,994	1,948,356	6.6	6,595	39,651	4.09	1.34
Money	org.totschnig.myexpenses	3.0.1.2	11.0	910.6	31.6	28.8	72.9	17.9	4.1	159,482,692	1,533,402	104.0	7,583	49,693	2.71	0.09
	com.igisw.openmoneybox	3.2.2.10	2.1	581.6	24.3	24.0	56.8	12.1	4.7	154,478,524	2,246,178	68.8	4,139	33,830	25.70	1.07
Multimedia	com.poupa.vinylmusicplayer	0.20.1	4.4	32.3	6.4	5.0	8.6	2.8	3.1	4,029,184	143,505	28.1	2,192	7,036	6.13	1.22
	org.gateshipone.odyssey	1.1.17	2.5	12,727.9	135.3	94.0	OOM	26.7	-	858,074,482	4,232,587	202.7	16,325	131,838	1.02	0.01
Navigation	com.ilm.sandwich	2.2.4f	3.1	6.9	2.5	2.7	1.0	**1.4**	0.7	690,044	45,712	15.1	1,485	3,189	8.99	3.30
	com.vonglasow.michael.satstat	3.3	2.5	8.3	0.8	10.1	12.7	0.6	21.2	1,388,430	21,673	64.1	785	2,282	62.41	6.21
Phone&SMS	opencontacts.open.com.opencontacts	12	2.0	1.4	1.3	1.1	0.6	**1.1**	0.5	43,904	16,918	2.6	370	1,149	6.71	6.31
	com.github.yeriomin.dumbphoneassistant	0.5	0.3	4.1	1.4	2.9	4.0	2.1	1.9	443,527	68,756	6.5	359	2,204	12.11	4.23
Reading	nightlock.peppercarrot	1.0.1	4.2	0.6	0.2	2.9	1.0	0.4	2.5	26,660	2,012	13.3	300	425	37.19	**12.82**
	org.decsync.sparss.floss	1.13.4	2.1	7,943.5	698.5	11.4	OOM	42.2	-	515,547,655	26,471,883	19.5	9,542	66,115	0.19	0.02
Sci&Edu	com.ichi2.anki	2.8.4	8.3	19.0	8.8	2.2	3.4	**6.2**	0.5	2,724,681	221,191	12.3	7,420	14,741	15.77	7.27
	com.luk.timetable2	6.0.4	2.8	0.2	0.1	1.3	1.0	0.9	1.1	1,753	298	5.9	71	73	20.00	**15.51**
Security	com.kunzisoft.keepass.libre	2.5.0.0beta18	7.3	6.9	0.7	9.3	14.8	1.0	14.8	1,002,449	20,218	49.6	622	1,863	13.88	1.49
	eu.faircode.netguard	2.229	2.5	11,034.6	589.5	18.7	OOM	17.5	-	935,368,414	14,070,780	66.5	15,290	235,259	0.31	0.02
Sports&Health	org.openpetfoodfacts.scanner	2.9.8	6.0	18,001.5	492.1	36.6	113.1	70.7	1.6	210,096,647	13,678,491	15.4	11,036	111,064	0.21	0.01
	org.secuso.privacyfriendlyactivitytracker	1.0.5	2.3	1.6	0.5	3.0	2.1	1.1	1.9	122,544	13,950	8.8	647	1,818	13.26	4.43
System	dk.jens.backup	0.3.4-universal	6.2	8.8	6.0	1.4	2.5	2.0	1.2	789,556	149,990	5.3	456	2,329	1.31	0.90
	com.github.axet.callrecorder	1.6.44	5.0	720.9	41.6	17.3	37.8	**64.2**	0.6	87,534,226	1,708,053	51.2	5,309	12,226	2.53	0.15
Theming	org.materialos.icons	2.1	8.5	3.2	0.4	7.7	4.4	0.8	5.5	282,413	2,255	125.2	108	108	58.21	7.54
	org.adw.launcher	1.3.6	1.2	4.5	1.5	3.0	6.1	0.8	7.6	504,404	34,408	14.7	1,204	1,353	17.39	5.74
Time	name.myigel.fahrplan.eh17	1.33.16	2.0	24.5	6.8	3.6	8.1	2.2	3.7	3,600,853	135,206	26.6	1,933	5,874	6.45	1.78
	com.app.Zensuren	1.21	0.2	8.0	2.9	2.7	7.9	1.7	4.6	1,765,179	177,806	9.9	2,452	4,958	37.08	**13.56**
Writing	com.orgzly	1.7	4.7	9,482.5	1,700.6	5.6	OOM	49.7	-	733,494,539	31,728,506	23.1	38,715	545,207	0.16	0.03
	org.secuso.privacyfriendlytodolist	2.1	2.4	8.4	1.5	5.4	1.6	1.3	1.2	928,849	62,544	14.9	956	5,884	11.54	2.13
GooglePlay	com.nianticlabs.pokemongo	0.139.3	97.0	18,002.5	380.9	47.3	123.2	47.4	2.6	321,729,505	11,194,524	28.7	9,134	107,311	0.14	0.00
	com.microsoft.office.word	16.0.11425.20132	71.0	257.6	28.1	9.2	70.8	8.8	8.0	29,801,488	343,827	86.7	3,466	16,023	1.39	0.15
	com.microsoft.office.outlook	3.0.46	70.0	18,001.9	1,069.6	16.8	122.3	79.2	1.5	498,296,938	21,980,834	22.7	52,165	413,532	0.96	0.06
	com.adobe.reader	19.2.1.9183	61.0	603.3	78.2	7.7	46.9	26.9	1.7	72,006,192	1,715,234	42.0	21,671	41,509	2.13	0.28
	com.emn8.mobilem8.nativeapp.bk	5.0.10	11.0	10,228.3	1,451.9	7.0	OOM	66.9	-	947,632,966	19,234,574	49.3	34,870	321,775	0.15	0.02
	de.schildbach.oeffi	10.5.3-google	2.1	312.2	14.8	21.1	18.9	6.1	3.1	50,630,514	405,221	124.9	7,388	23,337	6.79	0.32

after 12,727.9 seconds after running out of memory, SPARSEDROID takes only 135.3 seconds to run it to completion.

For `opencontacts.open.com.opencontacts` (with the smallest speedup, 1.1x), `com.ghstudios.android.mhgendatabase` (1.3x), `com.luk.timetable2` (1.3x), and `net.ddns.mlsoftlaberge.trycorder` (1.4x), the performance benefits from our sparse analysis are small. These apps can be analyzed by both tools within 1.5 seconds.

For `org.openpetfoodfacts.scanner`, `com.nianticlabs.pokemongo` and `com.microsoft.office.outlook`, FLOWDROID times out for a 5-hour time budget given, but SPARSEDROID has finished analyzing these three apps in 8.2 mins, 6.3 mins, and 17.8 mins, respectively.

C. RQ2: Memory Requirements

As shown in Columns 8 – 10 of Table II (also plotted in Figure 8), FLOWDROID consumes more memory than SPARSEDROID for all the 40 apps except for the five marked in **bold font** in Column 9 (discussed below), In particular, FLOWDROID runs out of memory for the six apps marked with OOM in Column 8. In contrast, SPARSEDROID can finish analyzing all the 40 apps by consuming a maximum of 79.2GB memory. For each app, we consider the maximum amount of memory used. This is calculated by using Java Runtime APIs.

With the six OOM apps excluded, the memory usage ratios of FLOWDROID over SPARSEDROID range from 0.5x (`com.ichi2.anki`) to 21.2x (`com.vonglasow.michael.satstat`) with an average of 3.7x.

Let us examine the five apps for which SPARSEDROID uses more memory than FLOWDROID. For `de.k3b.android.contentproviderhelper`, `com.ilm.sandwich` and `opencontacts.open.com.opencontacts`, the extra amount of memory used is small. In each case, the total amount of memory used by SPARSEDROID is below 1.5GB. For `com.ichi2.anki` and `com.github.axet.callrecorder`, SPARSEDROID uses about twice memory as FLOWDROID, due to the extra space taken by the SCFG cache used.

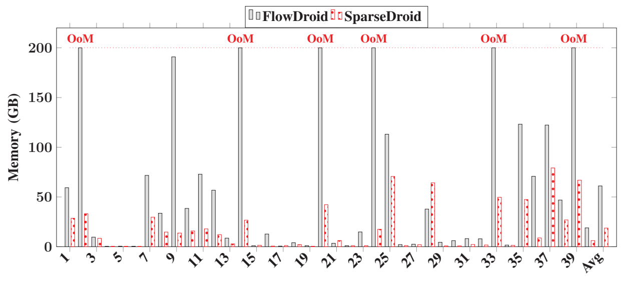


Fig. 8. Memory usage of FLOWDROID and SPARSEDROID for the 40 apps.

D. RQ3: Effectiveness of Sparse Analysis

As discussed in Sections V-B and V-C, SPARSEDROID is significantly faster and more memory-efficient than FLOWDROID. The key reason is that FLOWDROID, which uses the traditional IFDS algorithm in Figure 3, solves a significantly larger number of path edges (#PathEdges) than SPARSEDROID, which uses a sparse version in Figure 4, for an app, as shown in Columns 11 – 13 of Table II. The ratios of FLOWDROID's #PathEdges over SPARSEDROID's #PathEdges range from 2.6x to 385.5x with an average of 49.5x. As motivated in Figure 1, solving an IFDS problem sparsely cuts down substantially both the time and memory required.

In each IFDS algorithm, $WorkList$ contains all the path edges for an app to be solved by its IFDS solver. Figure 9 shows that $\frac{\text{FLOWDROID's Non-Sparse IFDS Solver Time}}{\text{SPARSEDROID's Sparse IFDS Solver Time}}$ correlates well positively with $\frac{\text{FLOWDROID}'s\ \#PathEdges}{\text{SPARSEDROID}'s\ \#PathEdges}$ across the 40 apps with both axes drawn in the $\log_2$ scale. The three blue ⊕'s represent the three apps for which FLOWDROID runs out of time and the six red ⊗'s represent the six apps for which FLOWDROID runs out of memory. The two blue ⊕'s (`org.openpetfoodfacts.scanner` and `com.nianticlabs.pokemongo`) deviate slightly from the projected trend, yielding better than expected speedups (Table II), possibly due to excessive system resources consumed by FLOWDROID.

Note that there is no correlation between the speedup of SPARSEDROID over FLOWDROID achieved for an Android app with its bytecode size. This is as expected since the effectiveness of SPARSEDROID depends on the tainted sources present in the app and their data dependent statements tracked (Figure 7) with aliases considered, which are all accurately reflected by the percentage of #PathEdges reduced (Figure 9).

When used as a vetting tool, SPARSEDROID can analyze all the 40 apps in 7678 seconds (2.13 hours) by issuing 228 leak warnings. By analyzing these apps (lexicographically) for the same time period, FLOWDROID can only analyze 30 apps by issuing only 147 leak warnings even if the nine apps for which FLOWDROID runs out of either time or memory are ignored.

E. RQ4: On-Demand SCFG Construction

As a sparse version of FLOWDROID, SPARSEDROID performs its taint analysis on-demand, by detecting the leaks from a set of specified sources to a set of specified sinks in an app. As such, our on-demand SCFG construction has turned out to be effective, by building the SCFGs for only the access paths encountered (Figure 1), as shown in Columns 14 – 17 of Table II. For each app, "#SCFGs" gives the number of SCFGs built and "#Acc Paths" gives the number of access paths seen (i.e., the number of SCFGs that would have been built if Theorem 2 were not applied). In addition, "Time/SD (%)" and "Time/FD (%)" give the percentages of the SCFG contruction time over the total analysis times spent by SPARSEDROID and FLOWDROID, respectively (plotted in Figure 10 graphically).

Certainly, SCFG construction incurs overheads, ranging from 0.14% to 62.4% with an average of 11.9% over SPARSEDROID's analysis time but only from 0.0% to 29.0% with an average of 3.3% over FLOWDROID's analysis time. As highlighted in **bold font** in the last Column of Table II, out of the six apps with double-digit percentage overheads (relative to FLOWDROID), `com.app.Zensuren` is the only one taking over 1 second for FLOWDROID to analyze. In this case, despite 1.08 seconds taken in building SCFGs, SPARSEDROID has reduced FLOWDROID's analysis time from 8 seconds to 2.9 seconds, achieving still a speedup of 2.7x. Overall, the overheads incurred in SCFG construction are significantly more than offset by the performance benefits reaped.

VI. LIMITATIONS

This work can be further improved in a number of directions. First, just like the traditional IFDS algorithm [1], our sparse IFDS algorithm (Figure 4) is applicable only to the IFDS data-flow problems. Second, by making FLOWDROID [19] sparse, our tool SPARSEDROID (Figure 6) is also limited to detecting only the information leaks caused by explicit data flows (via data-dependent assignments). How to track the sensitive information flowing implicitly through control-dependent assignments (i.e. `if (H) then L := true; else L := false`) [38], [39] is beyond the scope of this paper. Third, on-demand SCFG construction may introduce performance penalties for small CFGs. However, by applying this to all the CFGs in an app, the performance benefit seems to significantly more than offset the overheads incurred. Finally, the findings reported in this paper may be dependent on the set of Android apps selected.

VII. RELATED WORK

IFDS data-flow analyses are widely used in software testing, program verification, program understanding and maintenance, and compiler optimization. Reps et al. [1] initially introduced an efficient framework for solving the IFDS problems and subsequently generalized it to the IDE framework [32] for interprocedural distributed environment problems, where the dataflow facts are maps ("environments") from some finite set of symbols to some (possibly infinite) set of values. Later, Naeem et al. [22] give several extensions, making it applicable to a wider class of interprocedural data-flow problems, and also introduced a concurrent alternative implemented based on Scala's actor framework. WALA [40] contains a memory-efficient bit-vector-based IFDS algorithm. Recently, Bodden [23] has provided a generic (multi-threaded)

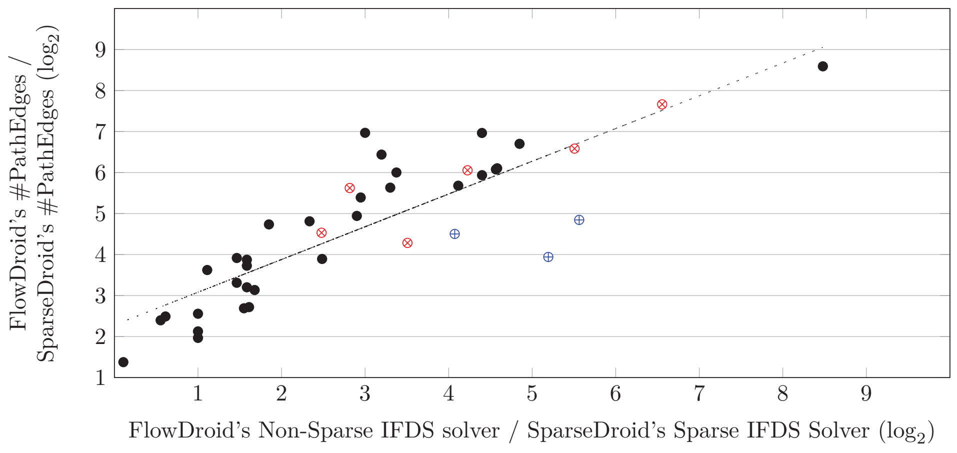


Fig. 9. Correlating #PathEdges with their solving time in an IFDS framework.

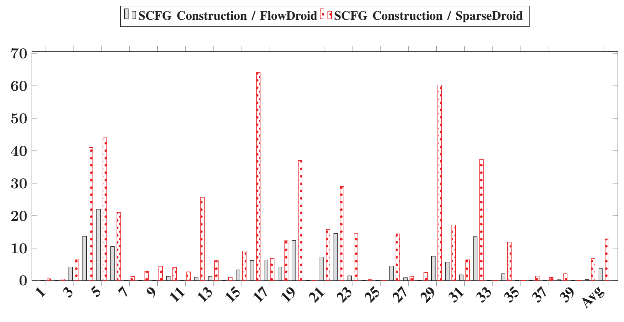


Fig. 10. Percentage of the time spent by SPARSEDROID on building SCFGs on-demand over the total analysis time by each tool for an app.

implementation of a generic IFDS/IDE solver in Soot [36]. While the prior work [22], [23] takes advantage of multi-threading to accelerate IFDS analyses, this work exploits their sparsity to improve their performance in an orthogonal way.

Sparse analysis has also been successfully applied in pointer analysis. Hardekopf et al. speed up flow-sensitive pointer analysis for C by employing a sparse data-flow graph rather than a dense control flow graph, initially for top-level variables [41] and then for also address-taken variables [42]. Sui et al. detect memory leaks in C programs by using a sparse value-flow graph that captures def-use chains and value flows via assignments for all memory locations represented by both top-level and address-taken variables [43] and also perform demand-driven flow- and context-sensitive pointer analysis for C programs sparsely [44]. Unlike these earlier efforts (focussing on standard pointer analysis algorithms), this paper represents the first work for sparsifying the *IFDS* algorithm, by constructing sparse CFGs *on-demand* instead of during a pre-analysis in order to boost its performance significantly.

Recently, synchronized pushdown systems are investigated as an alternative to the traditional storeless k-limited access path model for supporting pointer and data-flow analysis [45]. Their pushdown systems contain many redundant rules, which can be sparsified similarly to achieve performance gains.

Taint analysis is a form of data-flow analysis aiming at secure information flow. Given Android's popularity, many taint analysis tools exist, including Amandroid [46], DidFail [47], DroidSafe [48], EvoTaint [49], FLOWDROID [19] and IccTA [20], among which FLOWDROID remains to be a state-of-the-art static taint analysis tool [50]. However, FLOWDROID, with its taint analysis performed in an IFDS/IDE framework [23], is still compute- and memory-intensive [24], [51]. By sparsifying the traditional IFDS algorithm used, the performance of FLOWDROID has been significantly improved.

VIII. CONCLUSION

We have introduced a sparse analysis to scale the IFDS algorithm significantly by reducing its time and memory requirements. We have demonstrated that our sparse IFDS algorithm can improve substantially the scalability of taint analysis, one of the most important interprocedural data-flow analyses, on a range of Android apps. By constructing sparse CFGs on the fly for the data-flow facts propagated in both the forward taint analysis and backward alias analysis phases in the sparse IFDS framework, we have observed significant performance improvements. In future work, we plan to further reduce memory space consumed by taint analysis. We will also apply our sparse IFDS algorithm to other data-flow analyses.

ACKNOWLEDGEMENT

We would like to thank all the reviewers for their valuable inputs. This work is supported by the National Key R&D program of China (No. 2016YFB1000201), the Innovation Research Group of National Natural Science Foundation of China (No. 61521092), the National Natural Science Foundation of China (U1736028 and 61872043), and Australian Research Council Grants (DP170103956 and DP180104069).

REFERENCES

[1] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[2] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *International SPIN Workshop on Model Checking of Software*, 2000, pp. 113–130.

[3] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 232–241.

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[5] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *International Conference on Integrated Formal Methods*, 2004, pp. 1–20.

[6] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 57–68.

[7] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, p. 9, 2008.

[8] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 405–416.

[9] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program Tailoring: Slicing by Sequential Criteria," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 15:1–15:27. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6109

[10] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 56, 2016.

[11] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007, pp. 47–54.

[12] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[13] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "PSE: Explaining program failures via postmortem static analysis," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004, pp. 63–72.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 69–82.

[15] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 171–180.

[16] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android," Tech. Rep., 2009.

[17] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis," in *Presented as part of the 22nd {USENIX} Security Symposium*, 2013, pp. 543–558.

[18] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 87–97.

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[20] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 280–291.

[21] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable JavaScript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 177–187.

[22] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the IFDS algorithm," in *International Conference on Compiler Construction*, 2010, pp. 124–144.

[23] E. Bodden, "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*, 2012, pp. 3–8.

[24] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 426–436.

[25] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 598–608.

[26] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[27] (2019) Fossdroid. [Online]. Available: https://fossdroid.com/

[28] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," Ph.D. dissertation, 1991.

[29] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1973, pp. 194–206.

[30] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in SSA form," in *International Conference on Compiler Construction*, 1996, pp. 253–267.

[31] J. Späth, K. Ali, and E. Bodden, "Ideal: Efficient and precise alias-aware dataflow analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 99, 2017.

[32] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, vol. 167, no. 1-2, pp. 131–170, 1996.

[33] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 167–177.

[34] (2019) DroidBench: an open test suite for evaluating the effectiveness of taint-analysis tools specifically for Android apps. [Online]. Available: https://github.com/secure-software-engineering/DroidBench

[35] (2019) FlowDroid. [Online]. Available: https://github.com/secure-software-engineering/FlowDroid

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[37] (2019) Free and open source Android app repository. [Online]. Available: https://f-droid.org

[38] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *International Conference on Information Systems Security*, 2008, pp. 56–70.

[39] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and android intents," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 669–679.

[40] (2019) WALA: T.J. Watson Libraries for Analysis. [Online]. Available: http://wala.sourceforge.net/

[41] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009, pp. 226–238.

[42] ——, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011, pp. 289–298.

[43] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 254–264.

[44] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 460–473.

[45] J. Spth, "Synchronized pushdown systems for pointer and data-flow analysis," Ph.D. dissertation, University of Paderborn, Germany, 2019.

[46] F. Wei, S. Roy, X. Ou *et al.*, "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*, 2014, pp. 1329–1341.

[47] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "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*, 2014, pp. 1–6.

[48] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in DroidSafe," in *NDSS*, vol. 15, 2015, p. 110.

[49] H. Cai and J. Jenkins, "Leveraging historical versions of Android apps for efficient and precise taint analysis," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 265–269.

[50] F. Pauck, E. Bodden, and H. Wehrheim, "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*, 2018, pp. 331–341.

[51] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for Android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 106–117.