Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection

Dongjie He
University of New South Wales
Sydney, Australia
dongjieh@cse.unsw.edu.au

Yujiang Gui
University of New South Wales
Sydney, Australia
yujiang.gui@unsw.edu.au

Yaoqing Gao
Huawei Toronto Research Center
Toronto, Canada
yaoqing.gao@huawei.com

Jingling Xue
University of New South Wales
Sydney, Australia
jingling@cse.unsw.edu.au

ABSTRACT

The IFDS algorithm can be both memory- and compute-intensive for large programs as it needs to store a huge amount of path edges in memory and process them until a fixed point. In general, an IFDS-based data-flow analysis, such as taint analysis, aims to discover only the data-flow facts at some program points. Maintaining a huge amount of path edges (with many visited only once) wastes memory resources, and consequently, reduces its scalability and efficiency (due to frequent re-hashings for the path-edge data structure used).

This paper introduces a fine-grained garbage collection (GC) algorithm to enable (multi-threaded) IFDS to reduce its memory footprint by removing non-live path edges (i.e., ones that are no longer needed for establishing other path edges) from its path-edge data structure. The resulting IFDS algorithm, named Fpc, retains the correctness, precision, and termination properties of IFDS while avoiding re-processing GC'ed path edges redundantly (in the presence of unknown recursive cycles that may be formed in future iterations of the analysis). Unlike CleanDroid, which augments IFDS with a coarse-grained GC algorithm to collect path edges at the method level, Fpc is fine-grained by collecting path edges at the data-fact level. As a result, Fpc can collect more path edges than CleanDroid, and consequently, cause fewer re-hashings for the path-edge data structure used. In our evaluation, we focus on applying an IFDS-based taint analysis to a set of 28 Android apps. Fpc can scalably analyze three apps that CleanDroid fails to run to completion (under a 3-hour budget per app) due to out-of-memory (OoM). For the remaining 25 apps, Fpc reduces the number of path edges and memory usage incurred under CleanDroid by 4.4× and 1.4× on average, respectively, and consequently, outperforms CleanDroid by 1.7× on average (with 18.5× in the best case).

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598041

KEYWORDS

IFDS, Taint Analysis, Path Edge Collection

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598041

1 INTRODUCTION

The IFDS algorithm [36] addresses an important class of interprocedural data-flow analysis problems, in which the set of data-flow facts D is finite, the data-flow functions (in $2^D \mapsto 2^D$) associated with the control-flow edges in the program are distributive over the meet operator $\sqcap$, and the set of data-flow facts at each program point is a subset of D. It has been implemented in many mainstream program analysis and compiler frameworks [9, 23, 41] and used to solve a wide range of problems, such as taint analysis [2, 24], bug detection [15], pointer analysis [16, 18, 40], and typestate-like analysis [34, 44].

The IFDS algorithm is both memory- and compute-intensive, with a worst-case complexity of $O(|E| \cdot |D|^2)$ in space and $O(|E| \cdot |D|^3)$ in time , where E (the set of edges in the supergraph, i.e., a form of inter-procedural control flow graph of the program) and D (the set of data-flow facts in the program) are often large in practice. For example, a previous study [4] reported that FlowDroid (an IFDS-based taint analysis tool) [2] fails to analyze some Android apps on a computer server equipped with 730GB RAM (due to running out of either a 24-hour time budget or memory).

The IFDS algorithm solves an IFDS-based data-flow analysis as a graph reachability problem by expressing the data-flow facts reaching a program point in its supergraph as path edges. The objective of this paper is to introduce a garbage collection (GC) algorithm for IFDS to reduce its memory footprint and thus improve both its scalability and efficiency by garbage-collecting non-live path edges (i.e., ones that are no longer needed for establishing other path edges) from its path-edge data structure (e.g., a map in FlowDroid [2]). In general, IFDS-based data-flow analyses, such as taint analysis, aim to discover the data-flow facts at some program points. Maintaining all the path-edges in memory (several trillions for some apps in our evaluation), with many visited only once,

wastes memory resources, and consequently, reduces the scalability and efficiency of the IFDS analysis (due to frequent re-hashings for the path-edge data structure used [1]). Therefore, for the IFDS algorithm, garbage-collecting its non-live path edges will reduce not only its memory footprint but also its analysis time.

The challenge faced in this research is how to effectively garbage-collect non-live path edges for (multi-threaded) IFDS while both preserving its correctness, precision, termination properties and avoiding re-processing GC'ed path-edges redundantly, in the presence of unknown recursive cycles that may be formed (due to recursion or loops) in future iterations of the IFDS analysis. The only related work, CleanDroid [1], addresses this problem by augmenting IFDS with a coarse-grained method-level GC algorithm, by which the path edges in a method can be collected if these path edges and all their induced path edges in its transitively invoked callee methods have been processed. CleanDroid is precision-preserving, correct and safe (by guaranteeing termination), but not sufficiently efficient due to two limitations (as discussed in more details in Section 2). First, its method-level GC approach is overly conservative, failing to collect many non-live path edges during the IFDS analysis. Second, its GC algorithm may re-process some GC'ed path edges, resulting in redundant re-computations.

In this paper, we introduce a novel fine-grained GC approach for IFDS without these two limitations, by exploiting the structure of its path edges, which are always anchored at the start nodes of methods. The resulting IFDS algorithm is named Fpc. Compared with CleanDroid [1], Fpc is not only precision-preserving, correct and safe but also more efficient. We address CleanDroid's first limitation by collecting path edges at the data-fact level, based on a key observation that the path edges with different data facts at their sources, i.e., anchor sites can be GC'ed independently. We address CleanDroid's second limitation by introducing redundancy-avoiding edges to mark the GC'ed path-edges so that we can avoid re-processing them later completely.

To demonstrate the performance benefits of Fpc over CleanDroid (implemented in FlowDroid) [1], we have also selected taint analysis as a significant IFDS-based data-flow analysis as in [1] and implemented Fpc based on CleanDroid. We compare Fpc with CleanDroid on 28 Android apps (with 17 from [14] and 11 from [25]). Given a time budget (3 hours) and memory budget (200GB) per app, Fpc can scalably analyze three apps that CleanDroid fails to run to completion (due to OoM). For the other 25 apps, Fpc reduces the number of path edges and memory usage incurred under CleanDroid by 4.4× and 1.4×, respectively. As a result, Fpc speeds up CleanDroid by 1.7× on average (with 18.5× in the best case).

The paper makes the following main contributions:

- A novel fine-grained, data-fact-level GC algorithm for reducing the memory footprint of the IFDS algorithm, and consequently, improving its scalability and efficiency;
- Fpc, an open-source tool; and
- a performance evaluation of Fpc against CleanDroid.

The rest of this paper is organized as follows. Section 2 motivates our GC approach with an example. Section 3 formalizes our GC algorithm. We discuss the implementation of Fpc in Section 4 and evaluate Fpc against CleanDroid in Section 5. Finally, we discuss some related work in Section 6 and conclude the paper in Section 7.

2 MOTIVATION

We motivate our path-edge GC algorithm by considering an IFDS-based taint analysis. In Section 2.1, we give a motivating example and show how the IFDS algorithm works in detecting privacy leaks. Due to the technical nature of IFDS, we refer to [36] (Figure 4) for more details. In Section 2.2, we illustrate how CleanDroid [1] works by highlighting its two limitations. In Section 2.3, we introduce our GC approach without these two limitations.

2.1 A Motivating Example

Figure 1 gives a program containing three static methods, `main()`, `id()`, and `misc()` (in the gray-shaded background), where `id()` returns whatever value it receives and `misc()` simulates miscellaneous things irrelevant to privacy leaks. In `main()`, `a1.f` (line 5), `b1.g` (line 9), and `a2.f` (line 12) are tainted by *source()* immediately. When `a1` (`b1`) is passed into `id()`, `x.f` (`y.g`) becomes tainted at line 6 (line 10) and thus leaked at a *sink()* at line 8 (line 11). The calls to `misc()` (at lines 7 and 13) may produce more tainted access paths (e.g., `q.f`), which are assumed not to flow into any sink.

To detect privacy leaks in a program, the IFDS algorithm discovers the tainted access paths (as data-flow facts) reaching each program point by solving a graph reachability problem on its exploded supergraph [36]. The alias analysis used (orthogonal to this work) is discussed in Section 4. For a program, as reviewed in Section 3, its supergraph is just a form of inter-procedural CFG with call, return and call-to-return edges added, and its exploded supergraph, which is incrementally constructed, simply records each fact d reaching a node n (in the form of $\langle n, d\rangle$) and how each fact at a node propagates to taint the other facts at its successor nodes (in the form of $\langle n_1, d_1\rangle \rightarrow \langle n_2, d_2\rangle$). In Figure 1, we see the exploded supergraph for our example, where the special fact $\mathbf{0}$ allows new facts to be generated at `source()`. For example, the sets of facts reaching lines 8 and 11 are {$\mathbf{0}$, `a1.f`, `x.f`}, and {$\mathbf{0}$, `a1.f`, `x.f`, `b1.g`, `y.g`}, respectively. Thus, a leak at each of these two sinks is detected.

Given a program, IFDS builds its exploded supergraph implicitly by expressing it as a set of path edges [36] (Figure 4). For a method m, let s_m (e_m) be the start (exit) node in its own CFG. All its path edges must be anchored at the start node s_m. Whenever a new fact (taint) d_1 reaches m from a call site, a self-loop path edge $\langle s_m, d_1\rangle \rightarrow \langle s_m, d_1\rangle$ is first created. Then propagating d_1 to a node n in m may cause $\langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle$ to be created, meaning that d_1 at s_m causes d_2 at n to be tainted. A path edge is said to *induce* or *generate* or *establish* another path edge (directly or indirectly) if the former causes the latter to be created (directly or indirectly). By construction, $\langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle$ represents a suffix of a realizable path from $\langle s_{\texttt{main}}, \mathbf{0}\rangle$ to $\langle n, d_2\rangle$ in the exploded supergraph, meaning that starting at $\langle s_{\texttt{main}}, \mathbf{0}\rangle$, we will find first $\langle s_m, d_1\rangle$ and then $\langle n, d_2\rangle$ to be tainted. Let us see how IFDS detects that `x.f` is tainted after the call `x = id(a1)` at line 6. We write n_ℓ to represent the node at line ℓ. Initially, $\langle s_{\texttt{main}}, \mathbf{0}\rangle \rightarrow \langle s_{\texttt{main}}, \mathbf{0}\rangle$. At line 5, $\langle s_{\texttt{main}}, \mathbf{0}\rangle \rightarrow \langle n_5, \texttt{a1.f}\rangle$ is created. Due to the call `x = id(a1)` at line 6, `id()` is analyzed, with $\langle s_{\texttt{id}}, \texttt{p.f}\rangle \rightarrow \langle s_{\texttt{id}}, \texttt{p.f}\rangle$, $\langle s_{\texttt{id}}, \texttt{p.f}\rangle \rightarrow \langle n_{16}, \texttt{t.f}\rangle$, and $\langle s_{\texttt{id}}, \texttt{p.f}\rangle \rightarrow \langle e_{\texttt{id}}, \texttt{t.f}\rangle$ created. When $e_{\texttt{id}}$ is analyzed, $\langle s_{\texttt{main}}, \mathbf{0}\rangle \rightarrow \langle n_6, \texttt{x.f}\rangle$ is created in `main()`, indicating that `x.f` is tainted just after line 6.

Unfortunately, maintaining a huge amount of such path edges in memory introduces significant memory overheads without yielding

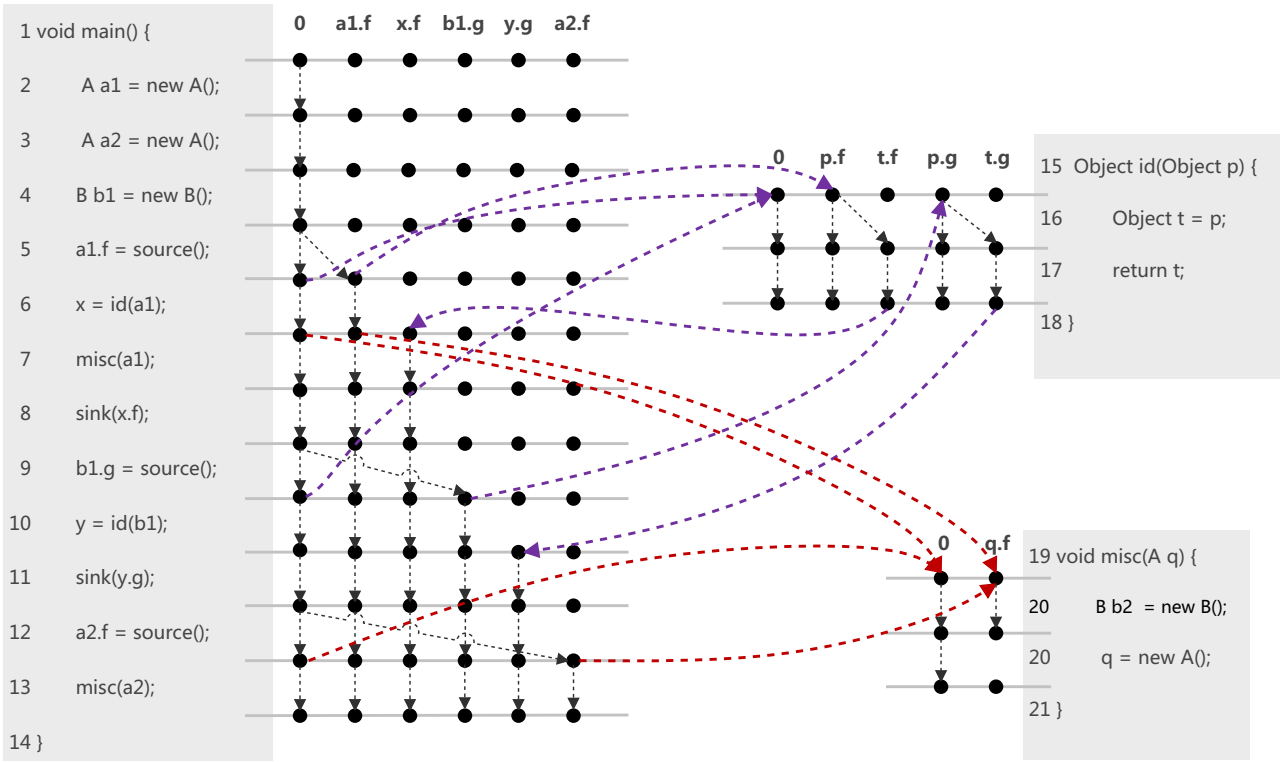


Figure 1: A motivating example and its exploded supergraph (implicitly) constructed during the IFDS analysis, where the inter-procedural edges (i.e., call and return edges) for method `id()` (`misc()`) are represented by ⇢ (⇢).

performance benefits since (1) most of them are visited only once [1, 25] (with 86.97% reported in [25]), and (2) the path edges recorded in memory are no longer used after the IFDS analysis (as all privacy leaks are detected on the fly). Thus, we can improve the scalability and efficiency of the IFDS algorithm if we can garbage-collect non-live path edges, i.e., ones that will no longer be used for establishing other path edges during the IFDS analysis, since removing non-live path edges from its path-edge data structure (e.g., a map in FlowDroid [2]) will reduce the number of re-hashings incurred [1].

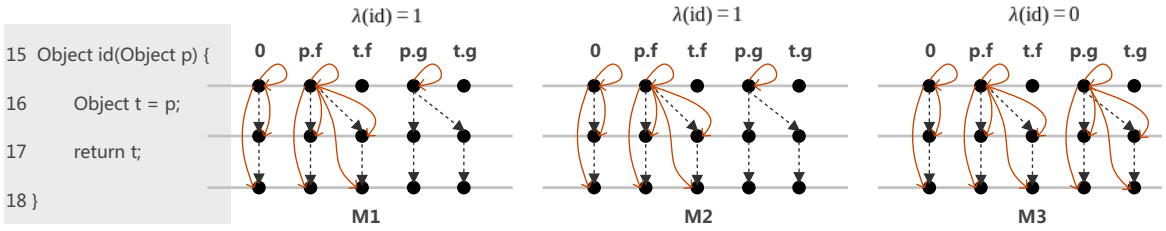
2.2 CleanDroid: Method-Level Collection

Recently, CleanDroid [1] represents an extension of the IFDS algorithm by garbage-collecting path edges during the IFDS analysis. The basic idea is simple. The path edges in a method can be collected if these path edges, together with all their induced path edges in its transitively invoked callee methods, have been processed. For example, the path edges in `id()` can be removed after line 10 and the path edges in `misc()` and `main()` can be removed after line 13.

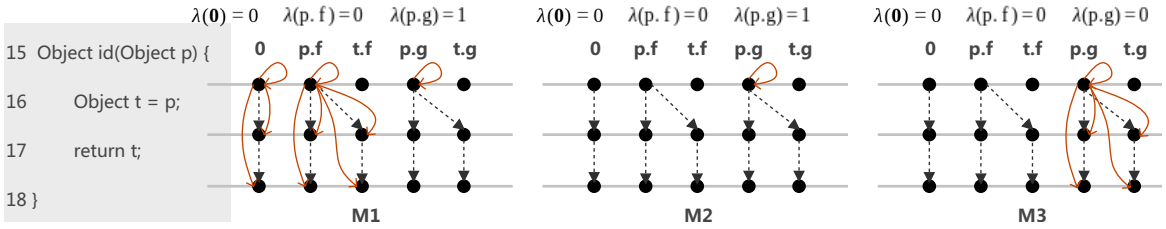
Let us consider just one GC point, at which the effect of the call `y = id(b1)` at line 10 is just about to be analyzed (i.e., the self-loop path-edge $\langle s_{\texttt{id}}, \texttt{p.g}\rangle \rightarrow \langle s_{\texttt{id}}, \texttt{p.g}\rangle$ is yet to be processed) but the effects of lines 1 – 9 have been analyzed. This implies that the effect of the call `x = id(a1)` at line 6 has been analyzed (i.e., the path edges starting (i.e., anchored) at $\langle s_{\texttt{id}}, \mathbf{0}\rangle$ and $\langle s_{\texttt{id}}, \texttt{p.f}\rangle$ in `id()` have been processed), and similarly, the effect of the call `misc(a1)` at line 7 has been analyzed. Let **M1** (**M2**) be the time just before (after) this GC point and **M3** the end of the analysis. For CleanDroid, Figures 2a and 3a give the snapshots of the path edges in `id()` and `misc()`, respectively, at **M1** – **M3**. We now examine its two limitations.

2.2.1 Limitation 1: Coarse Granularity. By using a method-level path-edge GC algorithm, CleanDroid misses many garbage collection opportunities due to its coarse granularity. As shown in Figure 2a, the path edges in `id()` at **M2** are the same as the ones at **M1**, implying that CleanDroid fails to collect any path edge for `id()` at this designated GC point. As a result, the maximum number of path edges maintained during the analysis is 13 at **M3**. CleanDroid has failed here since $\langle s_{\texttt{id}}, \texttt{p.g}\rangle \rightarrow \langle s_{\texttt{id}}, \texttt{p.g}\rangle$ is not yet processed (causing the reference counter of `id()` to be $\lambda(\texttt{id}) = 1 \neq 0$). Despite this unprocessed path edge, we observe that the other path edges in `id()`, anchored at $\langle s_{\texttt{id}}, \mathbf{0}\rangle$ and $\langle s_{\texttt{id}}, \texttt{p.f}\rangle$, have been processed and can thus be removed. Our fine-grained GC approach will address this limitation by collecting such path edges.

2.2.2 Limitation 2: Redundant Re-Computations. CleanDroid may suffer from redundant re-computations by re-processing some GC'ed path edges. As shown in Figure 3a, CleanDroid will remove the path edges created in `misc()` due to the first call to `misc()` at line 7 at the designated GC point (in between **M1** and **M2**) since the reference counter $\lambda(\texttt{misc}) = 0$ holds. However, when `misc()` is analyzed again due to the second call at line 13 , CleanDroid will end up re-introducing and re-processing these deleted path edges.

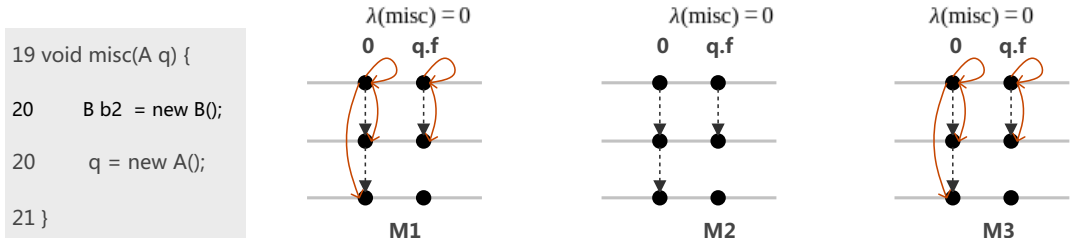


(a) The snapshots of the path edges in `id()` under CleanDroid

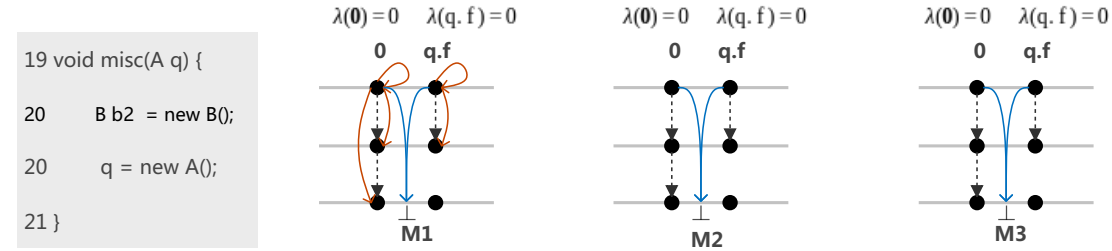


(b) The snapshots of the path edges in `id()` under Fpc ($\lambda(\langle s_{\mathtt{id}}, d\rangle)$ is abbreviated to $\lambda(d)$ for brevity)

Figure 2: Comparing how the path edges (depicted in orange arrows) change in `id()` under CleanDroid and Fpc with respect to a GC point when the path edges starting (i.e., anchored) at $\langle s_{\mathtt{id}}, \mathbf{0}\rangle$ and $\langle s_{\mathtt{id}}, \mathtt{p.f}\rangle$ have been processed but $\langle s_{\mathtt{id}}, \mathtt{p.g}\rangle \longrightarrow \langle s_{\mathtt{id}}, \mathtt{p.g}\rangle$ is yet to be processed. M1 (M2) represents the time just before (after) this collection point, and M3 the end of the IFDS analysis.



(a) The snapshots of the path edges in `misc()` under CleanDroid



(b) The snapshots of the path edges in `misc()` under Fpc ($\lambda(\langle s_{\mathtt{id}}, d\rangle)$ is abbreviated to $\lambda(d)$ for brevity)

Figure 3: Comparing the path edges ($\longrightarrow$) change in `misc()` under CleanDroid and Fpc, where M1, M2 and M3 are defined as in Figure 2. The redundancy-avoiding edges ($\longrightarrow$), once added, will not be removed during the IFDS analysis.

Thus, the path edges at **M1** and **M3** are identical. We will introduce redundancy-avoiding edges to avoid redundant re-computations.

2.3 Fpc: Data-Fact-Level Collection

We introduce a novel fine-grained GC approach that collects path edges at the data-fact level for IFDS without the two limitations of CleanDroid. The resulting IFDS algorithm is referred to as Fpc.

2.3.1 Fine Granularity. We address CleanDroid's first limitation by collecting path edges at the data-fact level instead of the method level. Given a method m, each of its path edges has the form of $\langle s_m, d_1\rangle \longrightarrow \langle n, d_2\rangle$, where s_m is its start node. Following [36], we refer to the source of $\langle s_m, d_1\rangle \longrightarrow \langle n, d_2\rangle$, i.e., $\langle s_m, d_1\rangle$ as its *anchor site*. Such a path edge is called a $\langle s_m, d_1\rangle$*-anchored path edge*.

We can collect path edges at the data-fact level by exploiting a key observation on the structure of path edges stated below.

Observation 1. *The path edges in a method with different anchor sites are handled independently during the IFDS analysis.*

This observation is always true since the IFDS analysis makes use of only distributive data-flow functions [36]. *As a result, for a given method, its path edges sharing the same anchor site can be*

garbage-collected if these path edges, together with their induced path edges in its transitively invoked callee methods, have been processed.

Let us return to our example by considering how Fpc collects the path edges in `id()`. Unlike CleanDroid, which fails to collect any path edge in our designated GC point (Figure 2a), Fpc has successfully removed all the path edges anchored at $\langle s_{\texttt{id}}, \mathbf{0}\rangle$ or $\langle s_{\texttt{id}}, \texttt{p.f}\rangle$ (Figure 2b), since they have been processed (indicated by $\lambda(\langle s_{\texttt{id}}, \mathbf{0}\rangle) = \lambda(\langle s_{\texttt{id}}, \texttt{p.f}\rangle) = 0$). As a result, the maximum number of path edges maintained at **M1** is 9 but drops to 1 at **M2**, and finally, reaches 5 at **M3**. Comparing Figure 2a and Figure 2b, we find that our fine-grained GC approach enables Fpc to maintain fewer path edges than CleanDroid (9 instead of 13) during the analysis.

Note that our approach works when `id()` contains directly or indirectly invoked callee methods contributing to the recursion cycles in the supergraph of the program, as formalized in Section 3.

2.3.2 Eliminating Redundant Re-Computations. We address CleanDroid's second limitation based on another key observation.

Observation 2. *For a given method m, all its path edges sharing the same anchor site $\langle s_m, d\rangle$ are generated directly or indirectly from a special self-loop path edge $\langle s_m, d\rangle \rightarrow \langle s_m, d\rangle$.*

This observation is always true due to how the path edges are constructed by the IFDS algorithm [36] (as presented in Figure 4).

As illustrated in Figure 3a, CleanDroid may re-process some GC'ed path edges. To avoid such redundant re-computations, whenever $\langle s_m, d\rangle \rightarrow \langle s_m, d\rangle$ is introduced (from a call site to m), we add a redundancy-avoiding edge $\langle s_m, d\rangle \rightarrow \bot$ to signify that all the $\langle s_m, d\rangle$-anchored path edges, once GC'ed, must have been processed, so that re-processing $\langle s_m, d\rangle \rightarrow \langle s_m, d\rangle$ from another call site can be avoided. As shown in Figure 3b, Fpc will collect the path edges anchored at $\langle s_{\texttt{misc}}, \mathbf{0}\rangle$ and $\langle s_{\texttt{misc}}, \texttt{q.f}\rangle$ (created due to the first call to `misc()` at line 7) at our designated GC point (in between **M1** and **M2**), since $\lambda(\langle s_{\texttt{id}}, \mathbf{0}\rangle) = \lambda(\langle s_{\texttt{id}}, \texttt{q.f}\rangle) = 0$), but will also add $\langle s_{\texttt{misc}}, \mathbf{0}\rangle \rightarrow \bot$ and $\langle s_{\texttt{misc}}, \texttt{q.f}\rangle \rightarrow \bot$ to prevent Fpc from re-processing these deleted path edges (due to the second call to `misc()` at line 13). Thus, the path edges at **M2** and **M3** are identical.

2.4 Discussion

Fpc is conceptually simple and can be added on top of a multi-threaded implementation of the IFDS algorithm in about 600 LOC. In addition, Fpc preserves the correctness, precision and termination properties of IFDS while avoiding re-processing GC'ed path edges. By collecting path edges at the data-fact level, Fpc can significantly improve the scalability and efficiency of CleanDroid.

3 OUR APPROACH

We describe our path-edge GC algorithm. We first review the classic IFDS algorithm [36] (Section 3.1). We then formalize our data-fact-level path-edge collector (Section 3.2), state a few important properties (Section 3.3), conduct an overhead analysis (Section 3.4), and finally, discuss some design choices (Section 3.5).

3.1 The IFDS Algorithm

The IFDS algorithm solves a special kind of data-flow problem, i.e., ***IFDS problem***. An instance of the IFDS problem, IP, is a quintuple, $IP = (G^*, D, F, M, \sqcap)$, where $G^* = (N^*, E^*)$ is the *supergraph* of the program, D is a finite set of data-flow facts, $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive data-flow functions, $M : E^* \mapsto F$ is a map from the supergraph edges to data-flow functions, and the meet operator $\sqcap$ is either union or intersection (depending on the problem modeled).

The supergraph of a program, G^*, consists of a set of CFGs, $G_1, G_2, \cdots$ (one per method), connected by inter-procedural edges. Given a method m, its CFG G_m has a *start* node s_m and an *exit* node e_m. A call site is represented by a *call* node and a *return* node. The other nodes (i.e., normal nodes) represent the statements as usual. The edges in G^* fall into four categories: *call edges* (connecting a call node to a start node), *return edges* (connecting an exit node to a return node), *call-to-return edges* (connecting a call node to a return node), and *normal edges* (connecting normal nodes).

Reps et al. [36] propose an efficient algorithm to solve the IFDS problem precisely by transforming it into a graph-reachability problem on an *exploded supergraph*, $G^\#_{IP} = (N^\#, E^\#)$, where $N^\# = N^* \times (D \cup \{\mathbf{0}\})$ and $E^\# = \{\langle n_1, d_1\rangle \rightarrow \langle n_2, d_2\rangle \mid n_1 \rightarrow n_2 \in E^*, f = M(n_1, n_2), d_2 \in f(d_1)\}$. Note that $f \in F$ is a data-flow function associated with the edge $n_1 \rightarrow n_2 \in E^*$. As mentioned earlier, $\mathbf{0}$ is a special fact used to introduce new facts at some program points.

In Figure 4, we give the IFDS algorithm [36] used in mainstream implementations [2, 7, 9, 38], where the lines shaded in light gray

```
1  Algorithm IFDS(G#_IP = (N#, E#))
2      InitPECollector()
3      PathEdge ← W ← S ← ∅
4      Propagate(⟨s_main, 0⟩ → ⟨s_main, 0⟩)
5      while W ≠ ∅ do
6          Pop ⟨s_m, d1⟩ → ⟨n, d2⟩ from W
7          if n is a call node then
8              Let m' be the method called at n and n' be the return node of n
9              for d3 such that ⟨n, d2⟩ → ⟨s_m', d3⟩ ∈ E# do
10                 ADG = ADG ∪ {⟨s_m, d1⟩ → ⟨s_m', d3⟩}
11                 Propagate(⟨s_m', d3⟩ → ⟨s_m', d3⟩)
12                 for ⟨s_m', d3⟩ → ⟨e_m', d4⟩ ∈ S ∧ ⟨e_m', d4⟩ → ⟨n', d5⟩ ∈ E# do
13                     Propagate(⟨s_m, d1⟩ → ⟨n', d5⟩)
14             for d3 such that ⟨n, d2⟩ → ⟨n', d3⟩ ∈ E# do
15                 Propagate(⟨s_m, d1⟩ → ⟨n', d3⟩)
16         if n is an exit node then
17             if ⟨s_m, d1⟩ → ⟨n, d2⟩ ∉ S then
18                 Insert ⟨s_m, d1⟩ → ⟨n, d2⟩ into S
19                 for each call site c that calls m do
20                     Let m'' (n'') be the containing method (the return node) of c
21                     for ⟨s_m'', d3⟩ → ⟨c, d4⟩ ∈ PathEdge ∧ ⟨c, d4⟩ →
                           ⟨s_m, d1⟩ ∈ E# ∧ ⟨n, d2⟩ → ⟨n'', d5⟩ ∈ E# do
22                         Propagate(⟨s_m'', d3⟩ → ⟨n'', d5⟩)
23         if n is a normal node or a return node then
24             for ⟨n', d3⟩ such that ⟨n, d2⟩ → ⟨n', d3⟩ ∈ E# do
25                 Propagate(⟨s_m, d1⟩ → ⟨n', d3⟩)
26         OnEdgeProcessed(⟨s_m, d1⟩ → ⟨n, d2⟩)
27 Procedure Propagate(⟨s_m, d1⟩ → ⟨n, d2⟩)
28     if ⟨s_m, d1⟩ → ⟨n, d2⟩ ∉ PathEdge then
29         if ⟨s_m, d1⟩ = ⟨n, d2⟩ then                    // a self-loop edge
30             if ⟨s_m, d1⟩ → ⊥ ∈ RAEdges then
31                 return // avoid redundant re-computations
32             else
33                 RAEdges = RAEdges ∪ {⟨s_m, d1⟩ → ⊥}
34         OnEdgeScheduled(⟨s_m, d1⟩ → ⟨n, d2⟩)
35         Insert ⟨s_m, d1⟩ → ⟨n, d2⟩ into both PathEdge and W
```

Figure 4: The IFDS algorithm [36] used in mainstream implementations [2, 7, 9, 38]. The lines shaded in light grey are used to support path-edge collection.

are added to support path-edge collection. *PathEdge* records the set of path edges, with each $\langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle$ representing the suffix of a realizable path (in $G^{\#}_{IP}$) from $\langle s_{\texttt{main}}, \mathbf{0}\rangle$ to $\langle n, d_2\rangle$. $\mathcal{S} \subseteq PathEdge$ represents the set of summary edges for summarizing the interprocedural data-flow facts obtained across the method boundaries. The IFDS algorithm is essentially a worklist algorithm. Starting from $\langle s_{\texttt{main}}, \mathbf{0}\rangle \rightarrow \langle s_{\texttt{main}}, \mathbf{0}\rangle$ (line 4), it removes one path edge from the worklist $\mathcal{W}$ and processes it at each iteration. Specifically, lines 7-15 handle the interprocedural data flows entering a method, lines 16-22 handle the interprocedural data flows leaving a method, and lines 23-25 handle the intraprocedural data flows within a method.

3.2 Data-Fact-Level Path-Edge Collection

Unlike CleanDroid [1], which can use the supergraph itself to perform its method-level path edge collection, Fpc relies on a new graph, named *Anchor Dependency Graph*, constructed during the IFDS analysis, to perform its data-fact-level path edge collection.

Definition 3.1. For a program, its *anchor dependency graph* is defined as $ADG = (N^{ADG}, E^{ADG})$, where a node $\alpha \in N^{ADG}$ is an anchor site and an edge $\alpha_1 \rightarrow \alpha_2 \in E^{ADG}$ means that some α_2-anchored path edges may be generated by some α_1-anchored path edges.

For a program, we build its *ADG* on the fly during the IFDS analysis. As shown in line 10 of Figure 4, whenever a self-loop path edge $\langle s_{m'}, d_3\rangle \rightarrow \langle s_{m'}, d_3\rangle$ is introduced (due to a call to a method m' made in the current method m being analyzed), we add $\langle s_m, d_1\rangle \rightarrow \langle s_{m'}, d_3\rangle$ to *ADG* to indicate that some $\langle s_{m'}, d_3\rangle$-anchored path edges may be generated by some $\langle s_m, d_1\rangle$-anchored path edges (i.e., $\langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle$ in line 6 of Figure 4).

Let $\lambda(\alpha)$ be the reference counter recording the number of α-anchored path edges that are being processed at the current iteration or will be processed in later iterations. Given an anchor site $\alpha \in N^{ADG}$, let $\texttt{ADGTC}(\alpha)$ be the set of anchor sites reflectively and transitively reachable from α on *ADG*. We identify non-live path edges in *PathEdge*, which can be collected by Fpc, as follows.

Definition 3.2. We can garbage-collect the set of all α-anchored path edges, which are deemed as being *non-live*, from *PathEdge* if

$$\forall \alpha' \in \texttt{ADGTC}(\alpha) : \lambda(\alpha') = 0 \tag{1}$$

If this condition holds, all path edges induced (directly or indirectly) by the self-loop path edge $\alpha \rightarrow \alpha$ have already been processed, and will no longer be used to generate other path edges.

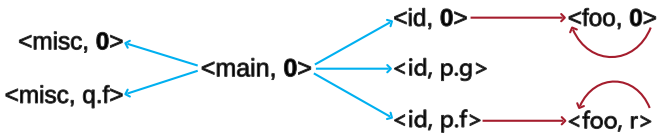


Figure 5: The *ADG* for Figure 1 (by adding a static method "`foo(Object r){foo(r);}`" and a call to `foo(p.f)` in `id()`).

Example 3.1. *For our motivating example, let us add a static method "*`void foo(Object r){ foo(r);}`*" and a call to* `foo(p.f)` *in* `id()`*. Figure 5 gives the ADG of this modified program (once fully analyzed). The only anchor site in* `main()` *is* $\langle$`main`, $\mathbf{0}\rangle$*. By Definition 3.1, after* `id()` *(called at lines 6 and 10) and* `misc()` *(called at lines 7 and 13) have been analyzed, we will have added the five edges (in cyan) to ADG. By analyzing the newly inserted code, we will introduce four more edges (in red). By definition,* $\texttt{ADGTC}(\langle \texttt{id}, \texttt{p.f}\rangle) = \{\langle \texttt{id}, \texttt{p.f}\rangle, \langle \texttt{foo}, \texttt{r}\rangle\}$*, and* $\texttt{ADGTC}(\langle \texttt{main}, \mathbf{0}\rangle) = N^{ADG}$ *since all the eight anchor sites in the modified program are reachable from* $\langle$`main`, $\mathbf{0}\rangle$.

```
1  Procedure InitPECollector()
2      λ = {α ↦ 0}
3      RAEdges = C = ∅
4  Procedure OnEdgeScheduled(⟨s_m, d_1⟩ → ⟨n, d_2⟩)
5      Consume(⟨s_m, d_1⟩ → ⟨n, d_2⟩) // do some analysis-specific task
6      λ = { α ↦ λ(α) + 1   if α = ⟨s_m, d_1⟩
           { α ↦ λ(α)       otherwise
7      C = C ∪ {α}
8  Procedure OnEdgeProcessed(⟨s_m, d_1⟩ → ⟨n, d_2⟩)
9      λ = { α ↦ λ(α) − 1   if α = ⟨s_m, d_1⟩
           { α ↦ λ(α)       otherwise
10 Procedure RunFineGrainedPECollector()
11     foreach α ∈ C do
12         if ∀α' ∈ ADGTC(α, ADG): λ(α') = 0 then
               // Remove path edges with α as their anchor site
13             foreach e : ⟨s_m, d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge do
14                 if α = ⟨s_m, d_1⟩ then
15                     PathEdge = PathEdge\{e}
16             C = C\{α}
```

Figure 6: Data-fact-level path-edge GC algorithm.

Figure 6 gives our data-fact-level path-edge GC algorithm. Let C be the set of candidate anchor sites found so far. For an anchor site in C, its anchored path edges will be removed from *PathEdge* once they satisfy Equation (1). Let $\mathcal{RAEdges}$ be the set of the redundancy-avoiding edges introduced so far. Initially, `InitPECollector()` is called (line 2 of Figure 4), in which we initialize the reference counters for all anchor sites to 0 and set $\mathcal{RAEdges} = C = \emptyset$ (lines 2-3). Whenever a new path edge $\langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle$ is ready to be scheduled, `OnEdgeScheduled()` is called (line 34 of Figure 4), in which we make the new path edge available via `consume()`, where, for example, privacy leaks are checked (line 5), increment the reference counter of $\langle s_m, d_1\rangle$ by 1 (line 6), and add $\langle s_m, d_1\rangle$ to C (line 7). Once a path edge has been processed, `OnEdgeProcessed()` is called (line 26 of Figure 4), in which we decrement the reference counter of its anchor site by 1 (line 9). Finally, `RunFineGrainedPECollector()` runs in a separate thread concurrently to the (multi-threaded) IFDS analysis itself, removing path edges from *PathEdge* as long as their anchor sites satisfy Equation (1) at each GC event (lines 11-16).

To avoid re-processing GC'ed path edges (as discussed in Section 2.2.2), we have made the following extension to the IFDS algorithm. For each self-loop path edge $\langle s_m, d_1\rangle \rightarrow \langle s_m, d_1\rangle$ that is ready to be scheduled (lines 29-33 of Figure 4), if $\langle s_m, d_1\rangle \rightarrow \bot$ is present in $\mathcal{RAEdges}$ (implying that the $\langle s_m, d_1\rangle$-anchored path edges, once GC'ed, must have already been processed), then nothing needs to be done. Otherwise, we add $\langle s_m, d_1\rangle \rightarrow \bot$ to $\mathcal{RAEdges}$.

3.3 Properties

Given an original IFDS analysis $\mathcal{A}$, let $\mathcal{A}'$ be $\mathcal{A}$ modified to incorporate our path-edge GC algorithm in Figure 6. Let $PathEdge_{\mathcal{A}}$ ($PathEdge_{\mathcal{A}'}$) be the set of path edges that have ever appeared in *PathEdge* and processed by $\mathcal{A}$ ($\mathcal{A}'$) at the end of the analysis.

Theorem 3.3 (Correctness and Precision). *For a program, $\mathcal{A}'$ achieves the same precision as $\mathcal{A}$ in the sense that both provide the same path edges to* `consume()` *(line 5 in* `OnEdgeScheduled()`*).*

Proof. For a program, $PathEdge_{\mathcal{A}} = PathEdge_{\mathcal{A}'}$ as both $\mathcal{A}$ and $\mathcal{A}'$ use the same data-flow functions. Whenever $e : \langle s_m, d_1\rangle \rightarrow \langle n, d_2\rangle \in PathEdge_{\mathcal{A}'}$ is removed, $\lambda(\langle s_m, d_1\rangle) = 0$ (line 13 in `RunFineGrainedPECollector()`). Thus, `OnEdgeScheduled()` has been called (implying that `consume(`e`)` has been called) and `OnEdgeProcessed()` has been called (implying that e has been processed). Thus, $\mathcal{A}'$ achieves exactly the same precision as $\mathcal{A}$. □

When a program contains recursive cycles (due to recursive calls or loops), some path edges may form a cycle, e.g., $e_0, e_1, \cdots, e_n$, where $e_{(i+1) \bmod (n+1)}$ is generated by e_i. To ensure termination, the original IFDS algorithm $\mathcal{A}$ uses a *PathEdge* data structure to maintain all path edges generated and thus prevents the same path edge from being re-generated and re-processed. However, a simple-minded path-edge GC algorithm may undermine its termination property, by repeatedly garbage-collecting a path edge and later adding it back again to *PathEdge*, especially when this deleted path edge appears in a path-edge-formed cycle that is not known at a GC event. Our GC algorithm is safe (by ensuring termination).

Theorem 3.4 (Safety). *For any given program, $\mathcal{A}'$ terminates.*

Proof. In the presence of recursive calls or loops in a given program, some path-edge-formed cycles, as discussed above, may exist. As $\mathcal{A}$ terminates, adding a simple-minded path-edge GC algorithm to $\mathcal{A}$ may cause $\mathcal{A}$ not to terminate only when it garbage-collects some but not all the path edges in one such a path-edge-formed cycle. For $\mathcal{A}'$, according to Equation (1), all α-anchored path edges are GC'ed conservatively only if these α-anchored path edges and all their induced path edges (including all the path-edge-formed cycles) have been processed. Thus, $\mathcal{A}'$ terminates since $\mathcal{A}$ terminates. □

Theorem 3.5 (No Redundant Re-Computations). *For a program, $\mathcal{A}'$ will never re-generate and re-process any GC'ed path edge, i.e., $\mathcal{A}'$ processes exactly the same set of path edges as $\mathcal{A}$.*

Proof. By Observation 2, all α-anchored path edges are generated (directly or indirectly) by the self-loop path edge $\alpha \rightarrow \alpha$ and will be removed by our path-edge GC algorithm (lines 13-17) once Equation (1) is satisfied. To avoid re-processing these α-anchored path edges, we only need to avoid re-generating and re-processing $\alpha \rightarrow \alpha$, which is guaranteed by the existence of the edge $\alpha \rightarrow \bot$ inserted into $\mathcal{RA}Edges$ (lines 29-33 of Figure 4). Thus, $\mathcal{A}'$ processes exactly the same set of path edges as $\mathcal{A}$. □

3.4 Path-Edge GC Overheads

We now investigate the extra time and space overheads introduced by $\mathcal{A}'$ on top of $\mathcal{A}$. We argue theoretically that our GC algorithm given in Figure 6 can deliver performance benefits for the IFDS algorithm despite some additional overheads introduced.

$\mathcal{A}'$ introduces some extra space overheads to $\mathcal{A}$ as $\mathcal{A}'$ needs to maintain *ADG* and $\mathcal{RA}Edges$. Let us conduct a worst-case analysis. The size of *ADG* is $O(|E^{call}| \cdot |D|^2)$, where $|E^{call}|$ ($< |E^*|$) is the number of call edges in the supergraph of the program. The size of $\mathcal{RA}Edges$ is $O(|M| \cdot |D|)$, where $|M|$ ($\leq |E^{call}| + 1 < |E^*|$) is the number of methods analyzed. Overall, the additional space overhead incurred is $O(|E^{call}| \cdot |D|^2)$, which is substantially smaller than $O(|E^*| \cdot |D|^2)$, i.e., $PathEdge_{\mathcal{A}'}$ (by noting that $PathEdge_{\mathcal{A}'} = PathEdge_{\mathcal{A}}$). Thus, once the path-edge reduction ratio exceeds $\frac{|E^{call}|}{|E^*|}$, $\mathcal{A}'$ can consume less memory than $\mathcal{A}$. This is evidenced by the significant memory usage reduced by Fpc over CleanDroid for relatively large apps as evaluated later (Section 5).

$\mathcal{A}'$ adds some extra time overheads to $\mathcal{A}$ due to the execution of the lines shaded in light gray in Figure 4 (for computing the path-edge collection metadata) and of `RunFineGrainedPECollector()` (for performing the actual path-edge collection). In Figure 4, line 10 takes $O(|E^{call}| \cdot |D|^2)$. Lines 2, 26, and 29–34 altogether take $O(|E^*| \cdot |D|^2)$ since the size of $PathEdge_{\mathcal{A}'}$ is $O(|E^*| \cdot |D|^2)$ with each path edge processed only once. Overall, the extra time overhead introduced by the lines shaded in light gray in Figure 4 is $O(|E^*| \cdot |D|^2)$. Let I be the number of times that `RunFineGrainedPECollector()` in Figure 6 is called during the IFDS analysis. The time spent on the I invocations of `RunFineGrainedPECollector()` is $O(I \cdot |E^{call}| \cdot |D|^2)$. For a total of I invocations, we can remove at most $|PathEdge_{\mathcal{A}'}|$, i.e., $O(|E^*| \cdot |D|^2)$ path edges at lines 13-15 and verify efficiently Equation (1) in $O(I \cdot |E^{call}| \cdot |D|^2)$ (where $|E^{call}| \cdot |D|^2$ is the size of *ADG*) at lines 11-12. I is determined by the GC interval used. Compared with the time complexity of the IFDS algorithm, which is $O(|E^*| \cdot |D|^3)$, the overall time overhead added, $O((|E^*| + I \cdot |E^{call}|) \cdot |D|^2)$, is often small. In fact, our path-edge GC algorithm will significantly reduce the amount of path edges maintained in *PathEdge* during the IFDS analysis, resulting in fewer reallocating and re-hashing operations on *PathEdge* and thus reducing the analysis time substantially.

3.5 Design Choices and Decisions

We discuss some design choices made when designing our path-edge GC algorithm. First, we have decided to use the summary edges maintained in $\mathcal{S}$ as in [1] without garbage-collecting them, since (1) $\mathcal{S}$ accounts for only a small amount of memory usage relative to *PathEdge* [25], (2) $\mathcal{S}$ is required to ensure the termination of IFDS, and (3) $\mathcal{S}$ is used to avoid redundant re-computations.

Second, the *ADG* for a program is incrementally constructed (via line 10 of Figure 4) and never GC'ed. Theoretically, once all the α-anchored path edges have been removed from *PathEdge*, all reachable anchor sites of α on *ADG* and their anchored path edges can be removed. We have decided not to do so since *ADG* is relatively small compared to $PathEdge_{\mathcal{A}'}$ (as evaluated in Section 5).

Finally, we can perform path-edge GC more aggressively while also ensuring termination by weakening Equation (1) to:

$$\lambda(\alpha) = 0 \tag{2}$$

Thus, all the α-anchored path edges in a method m can be removed from *PathEdge* as long as $\lambda(\alpha) = 0$. Note that $\lambda(\alpha) \neq 0$ may hold later since some new α-anchored path edges may be generated when the return edges of some callee methods in m are analyzed later, resulting in potentially some GC'ed α-anchored path edges to be re-generated and re-processed. We expected this aggressive version to reduce the memory usage of the IFDS algorithm further at the cost of some redundant re-computations. However, the resulting IFDS algorithm obtained turned out to be slower than Fpc without achieving any noticeable memory usage reduction.

4 IMPLEMENTATION

As CleanDroid [1] is built on top of FlowDroid (an IFDS-based taint analysis) [2] by incorporating its coarse-grained path-edge GC algorithm, we have implemented Fpc also in FlowDroid by incorporating our fine-grained path-edge GC algorithm for comparison purposes. Note that our path-edge GC approach is general and can be applied to all IFDS-based analyses. Specifically, we have implemented Fpc in FlowDroid (revision `d8c80ac`, where CleanDroid is already included) in about 600 lines of Java code.

To support our path-edge GC algorithm in Figure 6, we have designed a new IFDS solver, which is an adapted version of FlowDroid's IFDS solver (`FastSolver`), according to Figure 4. To enable updating a reference counter λ in a concurrent environment, we have followed CleanDroid's implementation by exploiting APIs such as `AtomicInteger` and `ConcurrentCountingMap`.

`RunFineGrainedPECollector()` given in Figure 6 can be run in multiple ways. For example, we can invoke it after line 6 in Figure 4 for every, say, 5000 iterations. In our implementation, Fpc adopts the same strategy as in CleanDroid [1], by invoking `RunFineGrainedPECollector()` at regular GC intervals.

Finally, FlowDroid (by default) uses a forward pass for discovering tainted access paths (during its taint analysis) and a backward pass for finding aliases (during its alias analysis). Both passes are IFDS-based analyses, running iteratively. For example, the forward IFDS solver can inject path edges to the backward IFDS solver for finding more aliases while the backward IFDS solver can also inject path edges to the forward IFDS solver for discovering more tainted access paths. To enable Fpc to collect path edges in this scenario, we adopt the same approach used in CleanDroid but in a more fine-grained way. Specifically, we run two path-edge garbage collectors, one per pass in a separate thread. We garbage-collect all the α-anchored path edges conservatively but safely when Equation (1) is satisfied in both the forward and the backward analyses.

Theorem 4.1. *For any given program, Fpc will report exactly the same privacy leaks as CleanDroid and FlowDroid.*

Proof. Follows from Theorem 3.3 and the fact that Equation (1) is strengthened above to handle the two passes in FlowDroid. □

5 EVALUATION

We demonstrate the significant performance benefits achieved by Fpc over CleanDroid when both are applied to perform the IFDS-based taint analysis on Android apps. As CleanDroid has been shown to be superior to FlowDroid in both memory usage and analysis time [1], it suffices to compare Fpc with CleanDroid in this paper. By Theorem 4.1, Fpc, like CleanDroid, achieves the same precision as FlowDroid. In addition, we have also validated the correctness of our implementation by using many benchmarks, including DroidBench [3] and open-source apps. Thus, we focus on evaluating the performance advantages of Fpc over CleanDroid.

Our evaluation addresses the following three research questions:

- **RQ1.** Can Fpc reduce the memory usage of CleanDroid?
- **RQ2.** Can Fpc reduce the analysis time of CleanDroid?
- **RQ3.** How does the performance of Fpc over CleanDroid vary under different path-edge GC intervals used?

Benchmark Selection. We have considered all the 58 apps used in [14] and [25] (with 40 from [14] and 18 from [25]), which have been carefully selected by their authors for evaluating the performance of their taint analysis tools against FlowDroid. We consider this set of apps to be suitable here, especially given the lack of standard benchmarks in the field. However, we have excluded 27 apps from this set, including (1) 9 apps, which cause FlowDroid (a newer version than that used in [14, 25]) to crash, (2) 13 apps, which are small (analyzable under 2 seconds by FlowDroid), making garbage-collecting their path edges unfruitful under a 1-second GC interval (for the same reason, we have excluded all apps in TaintBench[28]), and (3) 5 apps, which are unscalable with a 3-hour budget per app under either CleanDroid or Fpc (or FlowDroid) due to running out of memory (OoM) or out of time (OoT). In addition, we have also excluded 3 apps, `nya.miku.wishmaster`, `org.gateshipone.odyssey` and `com.github.axet.callrecorder` from [14] and kept their newer versions in [25]. Finally, we have settled with the remaining 28 apps (17 from [14] and 11 from [25]).

We cannot use the apps used for evaluating CleanDroid in [1], where 600 small apps were randomly selected from the Google Play Store, with 508 apps being analyzed to completion by FlowDroid under a 5-minute time budget per app. These apps are not publicly available (with no information about their names and versions being provided). It is worth mentioning that the speedups of CleanDroid over FlowDroid are reported to be up to 1.66× (with no average given but must be under 1.66×). Thus, the speedups of Fpc over CleanDroid reported here can be understood in this context.

Experimental Setting. Our experiments are performed on a Linux server, running Ubuntu 22.04.1 LTS (Jammy Jellyfish), with 8 CPU cores and 256GB RAM. We have set the maximum heap size of JVM as 200GB (with `-Xmx`) and left enough memory for the OS to execute other system services. For both Fpc and CleanDroid, as in [1], we allocate 8 threads to perform the IFDS-based taint analysis and 2 threads to perform path-edge collection (with one for the forward pass and one for the backward pass). The time budget is set as 3 hours per app. For all the other configurations, we have used their default values set in CleanDroid, including a 1-second path-edge GC interval for both CleanDroid and Fpc.

All data used are the average (geometric mean) of three runs.

Main Results. Table 1 gives the main results along three dimensions, the analysis time (Columns 4-5), the peak memory usage recorded during the analysis (Columns 6-7), and the maximum number of path edges recorded in *PathEdge* (i.e., the peak *PathEdge* usage), $|PathEdge|_{\max}$, during the analysis (Columns 8-9). We calculate the latter two every second during the analysis.

5.1 RQ1: Memory Usage

In Columns 6-7 of Table 1, we compare the peak memory usage of CleanDroid and Fpc over the set of 28 apps selected (with each memory usage reduction factor given in brackets). Fpc can analyze all these apps to completion without running out of memory (OoM). In contrast, CleanDroid runs OoM for three apps, `org.openpetfoodfacts.scanner`, `bus.chio.wishmaster` and `org.openpetfoodfacts.scanner`. In general, Fpc uses less memory than CleanDroid except for some small apps in the "≤ 3 mins"

Table 1: Comparing the performance of Fpc and CleanDroid. The apps are ordered in increasing order of CleanDroid's analysis time and divided into three groups: "≤ 3 mins", "> 3 mins", and unscalable apps. OoM stands for out of memory.

Group	APP	Version	Analysis Time (s)		Memory Usage (GB)		$\|PathEdge\|_{max}$ (K)	
			CleanDroid	Fpc	CleanDroid	Fpc	CleanDroid	Fpc
≤ 3 mins	com.github.yeriomin.dumbphoneassistant	0.5	4	4 (1.0×)	0.7	0.8 (0.9×)	4.1	2.5 (1.6×)
	org.csploit.android	1.6.5	4	4 (1.0×)	1.8	2.1 (0.9×)	0.3	0.1 (3.0×)
	com.ilm.sandwich	2.2.4f	5	4 (1.2×)	1.0	1.3 (0.8×)	2.5	1.1 (2.2×)
	com.kunzisoft.keepass.libre	2.5.0.0beta18	6	6 (1.0×)	2.4	2.7 (0.9×)	3.4	1.4 (2.5×)
	dk.jens.backup	0.3.4	6	6 (1.0×)	1.7	1.4 (1.2×)	3.9	2.1 (1.9×)
	org.gateshipone.odyssey	1.1.18	12	12 (1.0×)	1.2	1.2 (1.0×)	12.1	3.4 (3.6×)
	com.alfray.timeriffic	1.09.05	21	17 (1.2×)	3.7	2.7 (1.4×)	33.6	22.9 (1.5×)
	org.decsync.sparss.floss	1.13.4	31	33 (0.9×)	4.7	3.2 (1.5×)	21.5	8.2 (2.6×)
	com.github.axet.callrecorder	1.7.13	54	46 (1.2×)	7.1	5.4 (1.3×)	53.4	7.2 (7.4×)
	org.materialos.icons	2.1	55	33 (1.7×)	4.6	4.7 (1.0×)	121.0	54.6 (2.2×)
	com.app.Zensuren	1.21	57	47 (1.2×)	5.8	5.2 (1.1×)	55.1	31.1 (1.8×)
	name.myigel.fahrplan.eh17	1.33.16	66	47 (1.4×)	2.5	2.1 (1.2×)	36.6	6.5 (5.6×)
	com.emn8.mobilem8.nativeapp.bk	5.0.10	151	127 (1.2×)	2.0	2.7 (0.7×)	11.2	2.3 (4.8×)
> 3 mins	com.genonbeta.TrebleShot	1.4.2	207	175 (1.2×)	7.0	5.8 (1.2×)	54.7	22.5 (2.4×)
	com.microsoft.office.word	16.0.11425.20132	259	184 (1.4×)	5.0	4.8 (1.0×)	43.1	7.9 (5.5×)
	org.secuso.privacyfriendlytodolist	2.1	288	160 (1.8×)	17.0	14.7 (1.2×)	211.4	104.5 (2.0×)
	com.vonglasow.michael.satstat	3.3	316	218 (1.4×)	23.1	20.6 (1.1×)	200.2	146.1 (1.4×)
	com.igisw.openmoneybox	3.4.1.8	331	189 (1.8×)	20.0	13.2 (1.5×)	331.7	115.9 (2.9×)
	com.kanedias.vanilla.metadata	1.0.4	367	345 (1.1×)	22.7	13.4 (1.7×)	283.3	17.7 (16.0×)
	org.secuso.privacyfriendlyweather	2.1.1	388	21 (18.5×)	24.5	1.9 (12.8×)	463.0	5.4 (86.4×)
	com.adobe.reader	19.2.1.9183	416	191 (2.2×)	5.2	3.0 (1.7×)	8.0	0.8 (9.5×)
	org.totschnig.myexpenses	3.0.1.2	1904	146 (13.0×)	94.7	14.7 (6.5×)	1067.3	32.2 (33.1×)
	org.fdroid.fdroid	1.8-alpha0	1906	270 (7.1×)	12.6	5.1 (2.5×)	76.2	6.1 (12.5×)
	org.lumicall.android	1.13.1	2549	2111 (1.2×)	41.4	24.9 (1.7×)	366.5	49.0 (7.5×)
	nya.miku.wishmaster	1.5.0	8142	1852 (4.4×)	152.5	80.1 (1.9×)	2947.9	685.3 (4.3×)
Unscalable for CleanDroid	org.openpetfoodfacts.scanner	2.9.8	-	2629	OoM	85.9	-	1015.5
	bus.chio.wishmaster	1.0.2	-	5769	OoM	142.2	-	1542.2
	com.github.axet.bookreader	1.12.14	-	8154	OoM	104.5	-	3313.8
Geometric Mean	-	-	-	1.7×	-	1.4×	-	4.4×

group. On average, Fpc can reduce the peak memory usage of CleanDroid by 1.4× (excluding the three apps with OoM errors under CleanDroid). We can observe that Fpc can reduce memory usage more significantly for apps in the "> 3 mins" group than in the "≤ 3 mins" group, suggesting that Fpc works better for large apps. Overall, Fpc exhibits better scalability than CleanDroid.

This level of memory reduction obtained by Fpc over CleanDroid is attributed to a reduction in the maximum number of path edges, $|PathEdge|_{\max}$, maintained. As shown in Columns 8-9, Fpc achieves an average reduction factor of 4.4× over CleanDroid.

We have further compared Fpc and CleanDroid by correlating the reduction in $|PathEdge|_{\max}$ with the reduction in memory usage. As shown in Figure 7, the correlation coefficient between $\frac{\text{CleanDroid's } |PathEdge|_{\max}}{\text{Fpc's } |PathEdge|_{\max}}$ and $\frac{\text{CleanDroid's } Memory\ Usage}{\text{Fpc's } Memory\ Usage}$ computed by Excel's Data Analysis Tool provided in its Analysis Toolpak is 0.983, demonstrating a highly positive correlation. Note that $|PathEdge|_{\max}$ is not the only factor affecting memory usage. The data-flow facts actually consume a large portion of memory [1].

As analyzed in Section 3.4, the memory overhead introduced by *ADG* and $\mathcal{RAEdges}$ can often be negligible. Figure 8 compares the number of edges in *ADG* with the total number of path edges ever processed for each app. For all the 28 apps, the ratio of the former over the latter is less than 3.3%, with an average of 0.45%. Note that an *ADG* edge uses slightly less space than a path edge. In addition, we have also compared the size of $\mathcal{RAEdges}$ (containing our redundancy-avoiding edges) with the size of $\mathcal{S}$ (containing normal summary edges introduced by IFDS) and the total number of processed path edges. The average ratios, 18.9% and 0.61%, are also small.

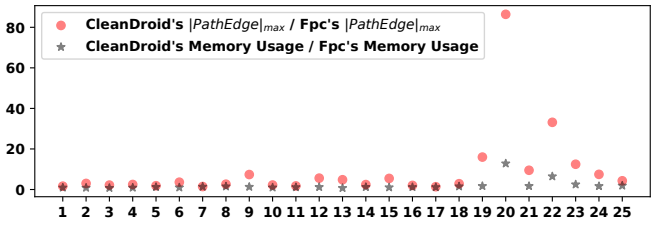


Figure 7: Comparing Fpc and CleanDroid by correlating the reduction in $|PathEdge|_{\max}$ with that in memory usage.

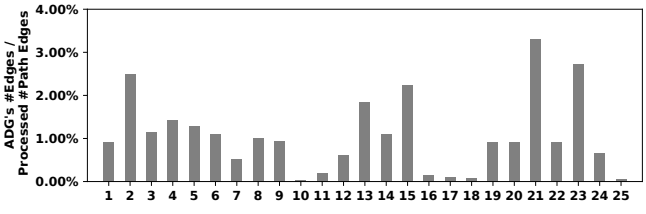


Figure 8: Comparing the number of edges in *ADG* with the total number of path edges ever processed (or generated).

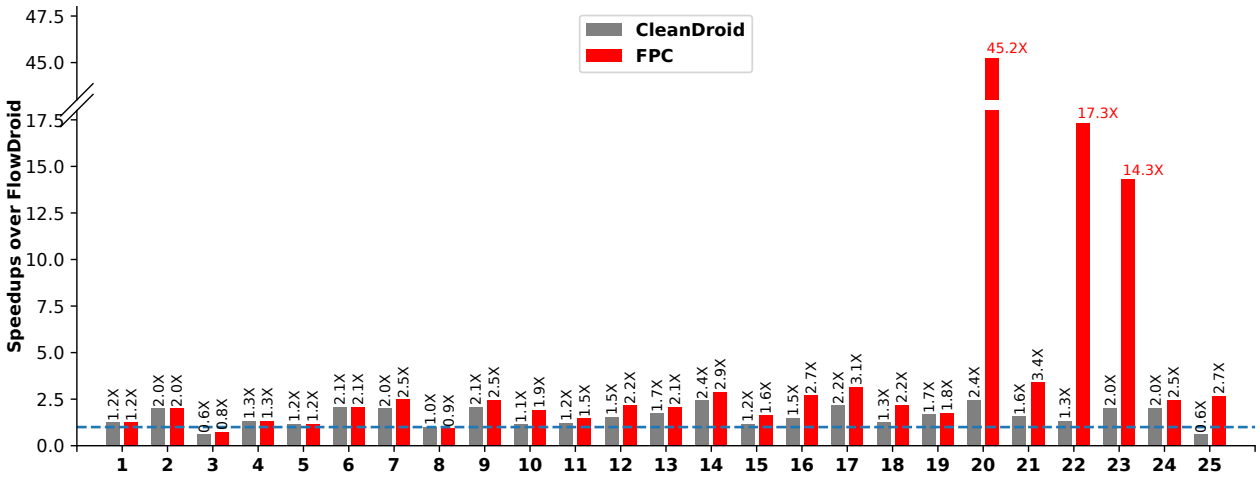


Figure 9: Comparing the speedups of Fpc and CleanDroid with FlowDroid as the baseline.

5.2 RQ2: Speedups

Garbage-collecting path edges from *PathEdge* (often implemented with a hash map) can reduce its load factor, resulting in fewer re-allocating and re-hashing operations during the analysis. Thus, both Fpc and CleanDroid can improve the performance of FlowDroid. In [1], CleanDroid is shown to outperform FlowDroid by up to 1.66× for a set of small Android apps. We show here that Fpc can boost the performance of CleanDroid significantly further.

As shown in Columns 4-5, Fpc can improve both the scalability and efficiency of CleanDroid. As discussed in Section 5.1, Fpc can analyze all the 28 apps scalably but CleanDroid runs OoM for 3 apps. For the remaining 25 apps, the speedups of Fpc over CleanDroid range from 0.9× to 18.5× with an average of 1.7×. In general, Fpc is more effective for large apps in the "> 3 mins" group, e.g., `org.secuso.privacyfriendlyweather` (18.5×), `org.totschnig.myexpenses` (13.0×), `org.fdroid.fdroid` (7.1×), and `nya.miku.wishmaster` (4.4×). For the "≤ 3 mins" group, Fpc is generally faster than CleanDroid but the speedups are relatively smaller.

Due to space limitations, we have not included the experimental results of FlowDroid, the standard IFDS-based taint analysis without GC, in Table 1. However, we can summarize the performance benefits achieved by comparing Fpc with CleanDroid using FlowDroid as the baseline. In the case of the last three apps listed in Table 1, both CleanDroid and FlowDroid cannot complete the analysis within the 3-hour time limit per app. On the other hand, Fpc is able to analyze these three apps in 2629s, 5769s, and 8154s, respectively. For the remaining 25 apps, as depicted in Figure 9, both CleanDroid and Fpc can improve the performance of FlowDroid. However, Fpc has proved to be significantly more effective. On average, CleanDroid outperforms FlowDroid by 1.5×, with a maximum speedup of 2.4× observed for `com.genonbeta.TrebleShot` and `org.secuso.privacyfriendlyweather`. In contrast, Fpc outperforms FlowDroid by an average of 2.6×, achieving a maximum speedup of 45.2× for `org.secuso.privacyfriendlyweather`.

We have investigated the reasons behind these performance speedups. Figure 10 compares Fpc and CleanDroid by correlating the reduction in $|PathEdge|_{\max}$ with that in analysis time. We can

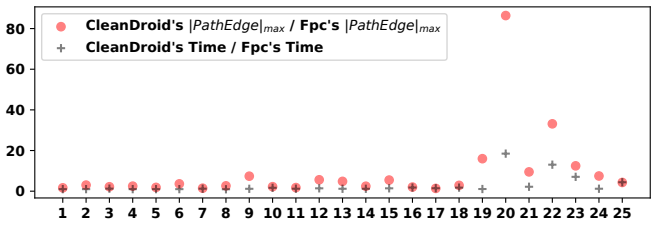


Figure 10: Comparing Fpc and CleanDroid by correlating the reduction in $|PathEdge|_{\max}$ with that in analysis time.

clearly observe a highly positive correlation (with the correlation coefficient being 0.922) between $\frac{\text{CleanDroid's } |PathEdge|_{\max}}{\text{Fpc's } |PathEdge|_{\max}}$ and $\frac{\text{CleanDroid's Analysis Time}}{\text{Fpc's Analysis Time}}$, meaning that a decrease in the former often leads to a decrease in the latter. Thus, performing GC at the data-fact level in Fpc instead of the method-level in CleanDroid is the major factor for the speedups achieved (Observation 1).

Despite not being a major contributing factor to the performance improvement of Fpc, we have included the avoidance of redundant path-edge re-processing (Observation 2) in Fpc for three reasons: First, it is theoretically significant, ensuring that Fpc processes the same set of path edges as the standard IFDS algorithm (Theorem 3.5). Second, it simplifies the complexity analysis of Fpc's overheads (Section 3.4). Finally, it has potential benefits for other IFDS instantiations beyond taint analysis.

5.3 RQ3: Fpc under Varying GC Intervals

We have also investigated how the performance of Fpc over CleanDroid varies under eight different GC intervals, as shown in Figure 11. For the average memory usage consumed per app, Fpc has improved CleanDroid by an average of 1.40 × ±0.03, showing that Fpc can reduce consistently the memory usage incurred by CleanDroid by about 1.40× per app. For the average analysis time per app, Fpc outperforms CleanDroid by 1.74 × ±0.02, again consistently across these GC intervals. We can draw two conclusions from Figure 11: (1) the overheads of Fpc (in both time and space) are

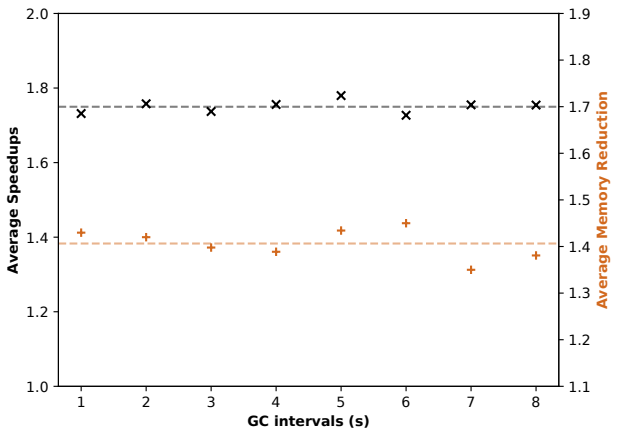


Figure 11: Improvements of Fpc over CleanDroid in terms of average memory usage and average analysis time per app across the intervals (i.e., with CleanDroid as the baseline).

negligible relative to the performance improvements achieved, and (2) the results obtained for **RQ1** and **RQ2** using the 1-second GC interval inherited from CleanDroid's default setting are reliable.

5.4 Threats to Validity

The performance speedups of Fpc over CleanDroid may vary across different IFDS-analyses subject also to the set of programs being considered. We have mitigated this major source of threats to validity by considering taint analysis as a significant IFDS-based analysis and evaluated Fpc against CleanDroid in FlowDroid [2] under the same setting by using a set of 28 Android apps from [14, 25]. In addition, we have conducted a time-and-space complexity analysis of our fine-grained path-edge GC algorithm in Section 3.4 by justifying theoretically why Fpc can improve the performance of the IFDS algorithm in general and of CleanDroid in particular (due to more path-edges that can be collected at the data-fact level). Our evaluation has also confirmed the performance advantages of Fpc over CleanDroid, particularly for large apps. To facilitate further research in this direction, Fpc is open-sourced.

6 RELATED WORK

We review the prior work closely related to the IFDS algorithm and the IFDS-based taint analysis for Android apps.

The IFDS algorithm was introduced initially by Reps et al. [36] for solving the IFDS problems and generalized subsequently to the IDE algorithm [37] for solving inter-procedural distributed environment problems. Later, Naeem et al. [35] gave several extensions to the IFDS algorithm, making it applicable to a wider class of inter-procedural data-flow problems. Recently, He et al. [14] have proposed a sparse IFDS algorithm that propagates data-flow facts directly to their next use points across the CFG of a method. Fpc is orthogonal to [14], implying that both can be combined together to achieve better performance benefits than either alone.

Currently, there are several popular implementations of the IFDS framework. Wala [9] contains a memory-efficient bit-vector-based implementation. There is a generic, multi-threaded implementation of the IFDS/IDE solver [7] in Soot [41]. In [38], a C/C++ implementation of the IFDS/IDE solver in Llvm [23] is also reported.

CleanDroid [1], which is the most related to our work, is the first to add a path-edge garbage collector to the IFDS algorithm. However, its method-level GC approach is coarse-grained and cannot avoid redundant path-edge re-processing. This paper proposes a novel fine-grained path-edge GC approach that can collect path-edges at the data-fact level without redundant re-computations.

Taint analysis can be either dynamic or static. Dynamic taint analysis [5, 10, 19–21, 29, 30, 39] tracks the flow of sensitive data during program execution. Static taint analysis, which represents a significant application of the IFDS algorithm, is widely used in providing secure information flow for Android apps. Many static taint analysis tools have emerged in the past few years, including Amandroid [43], DidFail [22], FlowDroid [2], IccTA [26], DroidSafe [11], P/Taint [12], and EvoTaint [8], of which FlowDroid (an IFDS-based framework) is the most popular. However, FlowDroid is both compute- and memory-intensive. To boost its performance, many techniques have been proposed, including sparse analysis [14], heap snapshots-assisted [6] and disk-assisted optimization [25, 42]. In [25], DiskDroid is proposed as a method to reduce the memory footprint of IFDS by swapping path edges between memory and disk. However, since DiskDroid is not open-sourced, a direct comparison with Fpc is not feasible. Nevertheless, according to the research paper, DiskDroid achieves only an 8.6% (1.086×) average speedup over FlowDroid. This speedup is significantly smaller than Fpc's performance improvement of 2.6× (as shown in Figure 9). Therefore, it can be concluded that Fpc is notably more effective than DiskDroid in enhancing the performance of IFDS. Like CleanDroid [1], Fpc, proposed in this paper, represents another technique focusing on memory usage reduction. In addition, some recent studies [27, 33], which investigate the precision and soundness of taint analysis tools, are orthogonal to our work.

Finally, IFDS [36] has found application in Qilin, a Java pointer analysis framework [17]. In Qilin, IFDS is employed in a pre-analysis [16, 18] to identify context-independent objects for context debloating in object-sensitive pointer analysis [31, 32].

7 CONCLUSION

We have introduced a new fine-grained path-edge GC algorithm for boosting the performance of the IFDS algorithm. Our approach can garbage-collect path edges at the data-fact level while avoiding re-processing GC'ed path-edges. To support IFDS-based taint analysis, we show that Fpc (FlowDroid augmented with our data-fact-level garbage collector) can improve substantially the scalability and efficiency of CleanDroid (FlowDroid augmented with a method-level garbage collector) on a set of Android apps.

DATA AVAILABILITY STATEMENT

The artefact of this paper [13] together with our implementation are publicly available at https://doi.org/10.5281/zenodo.7965678.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. This research is supported by ARC Grant DP210102409.

REFERENCES

[1] Steven Arzt. 2021. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 1098–1110.

[2] 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. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[3] Secure Software Engineering Group at Paderborn University and Fraunhofer IEM. 2022. DroidBench: an open test suite for evaluating the effectiveness of taint-analysis tools specifically for Android apps. Retrieved September 10, 2022 from https://github.com/secure-software-engineering/DroidBench

[4] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, New York, NY, USA, 426–436.

[5] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[6] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges, Eric Bodden, and Andreas Zeller. 2020. Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1061–1072. https://doi.org/10.1145/3377811.3380438

[7] 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*. Association for Computing Machinery, New York, NY, USA, 3–8.

[8] 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*. Association for Computing Machinery, New York, NY, USA, 265–269.

[9] IBM T.J. Watson Research Center. 2022. WALA: T.J. Watson Libraries for Analysis. Retrieved September 7, 2022 from http://wala.sourceforge.net/

[10] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490

[11] 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 *NDSS*, Vol. 15. 110.

[12] Neville Grech and Yannis Smaragdakis. 2017. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[13] 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 (Artifact). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*. https://doi.org/10.5281/zenodo.7965678

[14] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 267–279.

[15] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 167–177.

[16] Dongjie He, Jingbo Lu, and Jingling Xue. 2021. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 79–91. https://doi.org/10.1109/ASE51524.2021.9678880

[17] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[18] Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-Based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Transactions on Software Engineering and Methodology* (jan 2023). https://doi.org/10.1145/3579641

[19] Katherine Hough and Jonathan Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2, Article 26 (dec 2021), 43 pages. https://doi.org/10.1145/3485464

[20] Kaihang Ji, Jun Zeng, Yuancheng Jiang, Zhenkai Liang, Zheng Leong Chua, Prateek Saxena, and Abhik Roychoudhury. 2022. FlowMatrix: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2567–2584.

[21] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2018. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[22] 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*. Association for Computing Machinery, New York, NY, USA, 1–6.

[23] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, New York, NY, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[24] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient Context-Sensitive inside-out Taint Analysis for Large Codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 98–108. https://doi.org/10.1145/2635868.2635878

[25] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, New York, NY, USA, 236–247.

[26] 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*. IEEE, New York, NY, USA, 280–291.

[27] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 102–114. https://doi.org/10.1109/ASE.2019.00020

[28] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering* 27, 1 (2022), 1–41.

[29] Björn Mathis, Vitalii Avdiienko, Ezekiel O Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 263–273. https://doi.org/10.1109/ASE.2017.8115639

[30] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B Cohen, and Justyna Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 1358–1362. https://doi.org/10.1109/ASE51524.2021.9678758

[31] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. Association for Computing Machinery, New York, NY, USA, 1–11.

[32] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (2005), 1–41.

[33] Austin Mordahl and Shiyi Wei. 2021. The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 466–477. https://doi.org/10.1145/3460319.3464823

[34] Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like Analysis of Multiple Interacting Objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. Association for Computing Machinery, New York, NY, USA, 347–366. https://doi.org/10.1145/1449764.1449792

[35] Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical extensions to the IFDS algorithm. In *International Conference on Compiler Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

[36] 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*. Association for Computing Machinery, New York, NY, USA, 49–61.

[37] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1-2 (1996), 131–170.

[38] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for C/C++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer International Publishing, Cham, 393–410.

[39] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1527–1543. https://doi.org/10.1109/SP40000.2020.00022

[40] 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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[42] 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. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

[43] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 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*. Association for Computing Machinery, New York, NY, USA, 1329–1341.

[44] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327

Received 2023-02-16; accepted 2023-05-03