Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing

Haofeng Li[†‡], Haining Meng[†‡], Hengjie Zheng[†‡], Liqing Cao[†‡], Jie Lu[†], Lian Li[†‡*] and Lin Gao[§]
[†] State Key Laboratory of Computer Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China
[‡] University of Chinese Academy of Sciences, Beijing, China
[§] TianqiSoft Inc, China
[†] {lihaofeng19b, menghaining, zhenghenjie, caoliqing19s, lujie, lianli}@ict.ac.cn [§] gaolin@tianqisoft.cn

2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-7281-8613-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CGO51591.2021.9370311

***Abstract*—The IFDS algorithm can be memory-intensive, requiring a memory budget of more than 100 GB of RAM for some applications. The large memory requirements significantly restrict the deployment of IFDS-based tools in practise. To improve this, we propose a disk-assisted solution that drastically reduces the memory requirements of traditional IFDS solvers. Our solution saves memory by 1) recomputing instead of memorizing intermediate analysis data, and 2) swapping in-memory data to disk when memory usages reach a threshold. We implement sophisticated scheduling schemes to swap data between memory and disks efficiently.**

We have developed a new taint analysis tool, DiskDroid, based on our disk-assisted IFDS solver. Compared to FlowDroid, a state-of-the-art IFDS-based taint analysis tool, for a set of 19 apps which take from 10 to 128 GB of RAM by FlowDroid, DiskDroid can analyze them with less than 10GB of RAM at a slight performance improvement of 8.6%. In addition, for 21 apps requiring more than 128GB of RAM by FlowDroid, DiskDroid can analyze each app in 3 hours, under the same memory budget of 10GB. This makes the tool deployable to normal desktop environments. We make the tool publicly available at https://github.com/HaofLi/DiskDroid.

***Index Terms*—IFDS, taint analysis, memory consumption, scalability.**

I. Introduction

The IFDS (inter-procedural, finite, distribute, subset) analysis framework by Reps et al. [1] solves inter-procedural, context-sensitive, and flow-sensitive analysis of finite distribute subset problems where the set of data-flow facts D is finite and the transferring functions F (in $2^D \mapsto 2^D$) are distributive over the meet operator $\sqcap$ (either union or intersection). Such frameworks have been implemented in various analysis and compilation frameworks, including WALA [2], SOOT [3], and LLVM [4]. Those analysis frameworks are instrumental tools to solve a wide range of problems, such as pointer analysis [5], [6], [7], [8], taint analysis [9], [10], [11], [12], slicing [13], bug detection[14], [15], [16], [17], and shape analysis [18].

Existing implementations adopt the Tabulation algorithm [19], which solves a generalized graph-reachability problem (reachability along inter-procedurally realizable paths) in a super-graph extended from the inter-procedural control flow graph (CFG) of a program. Nodes in the super-graph are elements in the finite domain of data-flow facts at each program point (node $< n, d >$ denotes the data-flow element $d \in D$ at program point n), and edges E represent transferring functions. The data-flow fact d holds at n if and only if node $< n, d >$ is reachable from the start node $< s_0, \mathbf{0} >$ where s_0 is the entry point of the program. Intra-procedurally, it amounts to solve a general graph reachability problem. Inter-procedurally, the analysis requires that data-flow facts propagated from a callsite to an invoked callee function f (via a call edge to the CFG of f) can only return to the same callsite, i.e., context-sensitively by matching call and return edges.

The Tabulation algorithm has worst-case time complexity of $O(|E||D|^3)$ and space complexity of $O(|E||D|)$. It can be compute- and memory-intensive when the program size ($|E|$) or the domain of data-flow facts ($|D|$) is large. For example, the study [20] applied FlowDroid [9] (an IFDS-based taint analysis tool) to a set of 2,950 Android apps on a computer server with 730GB of RAM and 64 Intel Xeon CPU cores, 16 apps are unanalyzable because the IFDS solver of FlowDroid (with even many compromises made) takes more than 24 hours to finish and uses all the memory. The sparse IFDS algorithm [10] can significantly improve the run-time performance (by 22.0X) and memory footprints (by 3.7X) of the original IFDS algorithm. However, it still requires a memory budget of 220GB per app in its experiments.

The aim of this work is to scale up IFDS algorithms so that they can run on normal desktop environments with a memory budget of less than 16GB. Two memory-saving strategies are presented. The first recomputes intermediate analysis data instead of memorizing them in memory, and the second swaps in-memory data (generated by the IFDS algorithm) to disk when memory usages exceed a threshold. Efficient scheduling schemes are implemented to compensate for the extra cost in swapping data in and out of memory. Existing disk-based analyses [21], [22] leverage graph systems [23], [24], [25] to divide a large graph into chunks and process one chunk in memory at a time. On the contrary, we propose a disk-assisted approach to extend existing in-memory analysis algorithms with efficient swapping schemes.

We have implemented a disk-assisted IFDS solver and applied it to taint analysis for finding information leaks in

[*] Corresponding author.

Android apps. FlowDroid [9] is a state-of-the-art taint analysis tool based on a multi-threaded IFDS solver. We develop a new tool, DiskDroid, by extending FlowDroid with our disk-assisted IFDS solver. DiskDroid can successfully analyze large apps within a memory budget of 10GB, in contrast to FlowDroid which often requires more than 128GB of RAM. In addition, DiskDroid achieves a performance speedup of 8.6% over FlowDroid, despite given a much smaller memory budget. We make the tool available at https://github.com/HaofLi/DiskDroid.

This paper makes the following contributions:

- We present a new disk-assisted approach to scale up IFDS algorithms. The approach optimizes memory usages via recomputation and efficient disk swapping. These optimizations are applicable to both IFDS solvers and IDE solvers.
- We implement DiskDroid, a taint analysis tool based on our disk-assisted IFDS solver for detecting information leaks in Android apps. The tool is publicly available at https://github.com/HaofLi/DiskDroid.
- We have evaluated DiskDroid against FlowDroid on a set of 2,053 open source apps from F-Droid [26]. Among the 2,053 apps, 19 apps require 10GB to 128GB of RAM for FlowDroid and DiskDroid analyzes them under the memory budget of 10GB, with a slight performance improvement of 8.6%. For 21/162 apps which take more than 128GB of RAM by FlowDroid, DiskDroid can analyze them in 3 hours, given a memory budget of 10GB.

The rest of the paper is organized as follows. Section II overviews the classical IFDS algorithm and its application to taint analysis. Section III highlights the memory consumption problem of existing IFDS solvers. Section IV illustrates our disk-assisted approach. We evaluate the effectiveness and efficiency of our approach in Section V. Section VI reviews related work and Section VII concludes this paper.

II. Background

We review the classical Tabulation IFDS algorithm and study FlowDroid as an instantiation of the algorithm in solving taint analysis problems.

A. The Tabulation Algorithm

In the original formulation by Reps et al. [1], an instance *IP* of an IFDS problem is a five-tuple, $IP = (G^*, D, F, M, \sqcap)$, where $G^* = (N^*, E^*)$ is the inter-procedural CFG (ICFG) of the program, D is a finite set of data-flow facts, $F \subseteq 2^D \mapsto 2^D$ is a set of distributive functions, $M : E^* \mapsto F$ is a map from edges in the ICFG to data-flow functions, and the meet $\sqcap$ is either union or intersection.

The ICFG G^* consists of a collection of CFGs, G_0, G_1, G_2 ... (one per function), where G_0 represents the entry function of the program. By convention, each CFG G_p consists of a unique entry node s_p and a unique exit node e_p. A callsite is split into two nodes, a *Call* node and a *retSite* node. At a callsite, inter-procedural call edges connect the *Call* node to the entry node of its callee functions, and return edges connect exit nodes of callee functions to the *retSite* node. Thus, data-flow facts can propagate inter-procedurally via call and return edges.

Algorithm 1: The Tabulation IFDS Algorithm reproduced from [19], with some notational changes

```
Globals: PathEdge, WorkList, Incoming, EndSum, $\mathcal{S}$;
  Algorithm Tabulate($G^{\#}_{IP}$):
    Let $(N^{\#}, E^{\#}) = G^{\#}_{IP}$
    $PathEge \leftarrow \{< s_0, 0 > \to < s_0, 0 >\}$
    $WorkList \leftarrow \{< s_0, 0 > \to < s_0, 0 >\}$
    $\mathcal{S} \leftarrow \emptyset$
    ForwardTabulateSLRPs()
    for $n \in N^*$:
        $X_n \leftarrow \{d_2 \in D | \exists d_1 \in (D \cup \{0\})\}$ s.t.,
          $< s_{proc(n)}, d_1 > \to < n, d_2 > \in PathEdge\}$
Procedure Prop($e$):
    if $e \notin PathEdge$:
        Insert $e$ into $PathEdge$; Insert $e$ into $WorkList$
Procedure processCall($< s_p, d_1 > \to < n, d_2 >$):
    for $d_3$ s.t. $< n, d_2 > \to < s_{calledProc(n)}, d_3 > \in$
      $E^{\#}$:
        Prop($< s_{calledProc(n)}, d_3 > \to$
          $< s_{calledProc(n)}, d_3 >$)
        $Incoming[< s_{calledProc(n)}, d_3 >] \cup= < n, d_2 >$
        for $< e_p, d_4 > \in$
          $EndSum[< s_{calledProc(n)}, d_3 >]$:
            for $d_5$ s.t. $< e_p, d_4 > \to$
              $< retSite(n), d_5 > \in E^{\#}$:
                $\mathcal{S}^{\#} \cup= < n, d_2 > \to$
                  $< retSite(n), d_5 >$
    for $d_3$ s.t. $< n, d_2 > \to < retSite(n), d_3 > \in$
      $\{E^{\#} \cup \mathcal{S}\}$:
        Prop($< s_p, d_1 > \to < retSite(n), d_3 >$)
Procedure processExit($< s_p, d_1 > \to < n, d_2 >$):
    $EndSum[< s_p, d_1 >] \cup= < e_p, d_2 >$
    for $< c, d_4 > \in Incoming[< s_p, d_1 >]$:
        for $d_5$ s.t. $< e_p, d_2 > \to < retSite(c), d_5 >$
          $\in E^{\#}$:
            $\mathcal{S} \cup= < c, d_4 > \to < retSite(c), d_5 >$
            for $d_3$ s.t., $< s_{proc(c)}, d_3 > \to < c, d_4 >$
              $\in PathEdge$:
                Prop($< s_{proc(c)}, d_3 > \to$
                  $< retSite(c), d_5 >$)
Procedure ForwardTabulateSLRPs():
    while $WorkList \neq \emptyset$:
        Pop edge $< s_p, d_1 > \to < n, d_2 >$ from
          $WorkList$
        switch $n$:
            case $n \in Call_p$:
                processCall($< s_p, d_1 > \to < n, d_2 >$)
            case $n = e_p$:
                processExit($< s_p, d_1 > \to < n, d_2 >$)
            case $n \in (N_p - Call_p - \{e_p\})$:
                for $m, d_3$ s.t. $< n, d_2 > \to < m, d_3 >$
                  $\in E^{\#}$:
                    Prop($< s_p, d_1 > \to < m, d_3 >$)
```

To solve *IP* context-sensitively as a graph reachablility problem, G^* is extended to an exploded super-graph $G^{\#}_{IP} =$

$(N^{\#}, E^{\#})$ such that $N^{\#} = N^{*} \times (D \cup \{\mathbf{0}\})$ and $E^{\#} = \{< m, d_1 > \rightarrow < n, d_2 > — m \rightarrow n \in E^{*}, d_2 \in f(d_1)\}$. Note that here **0** signifies an empty set of facts such that new data-flow facts can be generated at a program point, and $f \in F$ is the flow function of the instruction at m. In the formulation, flow function f is replaced with a graph representation of $M(m \rightarrow n)$. For efficiency, $G^{\#}_{IP}$ is usually built from G^{*} during the analysis.

Algorithm 1 reproduces the Tabulatation IFDS algorithm [19]. In the algorithm, $Call_p$ is the set of call statements in function p, $calledProc(n)$ denotes the callee function invoked at call statement n and $retSite(n)$ is its corresponding return site. For a node $n \in G^{*}$, $proc(n)$ identifies its containing function.

The input of the algorithm is the super-graph $G^{\#}_{IP}$. The Tabulation algorithm maintains a set of data structures summarized below:

- *PathEdge* records the set of *path edges*, representing a subset of the same-level realizable paths in $G^{\#}_{IP}$, where the source is a node of the form $< s_p, d_1 >$ reachable from node $< s_0, \mathbf{0} >$. In other words, a path edge $< s_p, d_1 >$ to $< n, d_2 >$ represents the suffix of a realizable path from $< s_0, \mathbf{0} >$ to $< n, d_2 >$.
- $\mathcal{S}$ records the set of *summary edges*, which summarize inter-procedural data-flow facts across function boundaries, i.e., realizable paths from callsites to their corresponding return sites.
- *Incoming* records the set of nodes $< s_p, d >$ reachable from $< s_0, 0 >$, and their predecessors.
- *EndSum* records the set of path edges $< e_p, d_2 >$ from node $< s_p, d_1 >$.

The Tabulation algorithm is a worklist algorithm that accumulates sets of path edges and summary edges until a fixed point. Starting with $< s_0, \mathbf{0} > \rightarrow < s_0, \mathbf{0} >$ (lines 3 - 5), procedure `ForwardTabulateSLRPs` (line 6) collects all possible path edges in *PathEdge* with a case analysis (lines 31 - 38). There are 3 cases: 1) the inter-procedural data-flows entering into a function (lines 32 and 33), handled by procedure `processCall` (lines 12 - 20), 2) the inter-procedural data-flows leaving a function (lines 34 and 35), handled by procedure `processExit` (lines 21 - 27), and 3) the intra-procedural data-flows within a function (lines 36 - 38). Note that *Incoming* and *EndSum* are introduced in [19] to process inter-procedural flows more efficiently. *Incoming* is updated when entering into a function (line 15), and is queried when leaving a function (line 23). *EndSum* is updated when leaving a function (line 22), and is queried when entering into a function (line 16).

The algorithm terminates when no more path edge can be collected and the meet-over-all-valid-paths solution to IP at program point n is then computed as X_n (lines 7 and 8).

B. FlowDroid

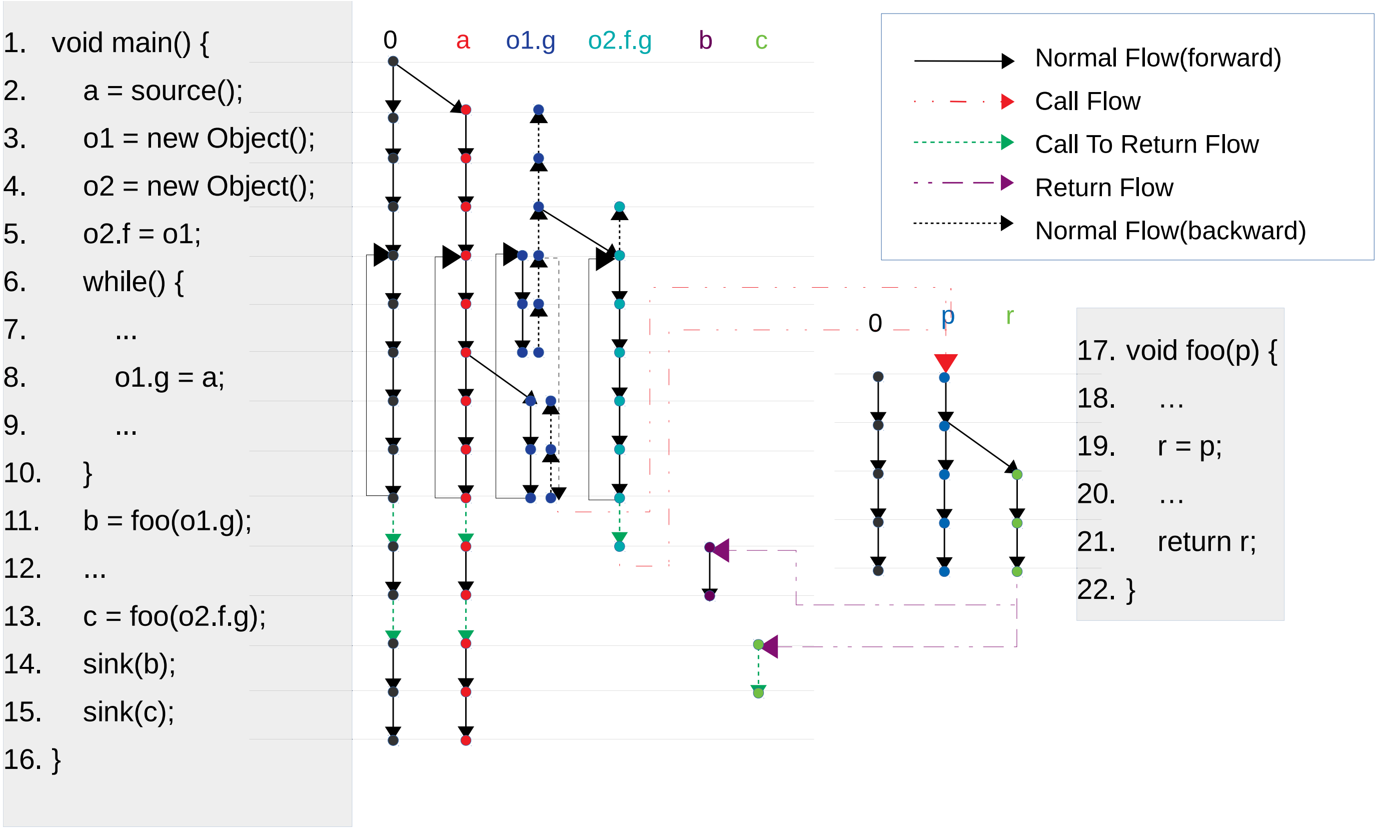
FlowDroid is a state-of-the-art IFDS-based taint analysis tool. Figure 1 depicts how the tool applies IFDS to solve taint analysis problems. Each node in the super-graph $G^{\#}_{IP}$ represents a data-flow fact (i.e., a tainted access path with fixed length) at a program point and edges propagate facts along the ICFG of the program. The meet operator $\sqcap$ is $\cup$ since an access path is regarded as tainted at a joint point if it is tainted in any of its incoming control-flow paths.

Data-flow facts are generated by assigning a tainted value to another (e.g., line 8 generates access path `o1.g` by assigning the tainted value `a` to `o1.g`), and killed by reset it to an untainted value. Those transferring functions are encoded as edges in $G^{\#}_{IP}$. For instance, the transferring function at line 8 is represented by the edge $< 8^{\bullet}, \texttt{a} > \rightarrow < 8_{\bullet}, \texttt{o1.g} >$, where $8^{\bullet}$ and $8_{\bullet}$ denote the program points before and after the statement at line 8, respectively. There are four types of edges: *normal* flow edges to propagate data-flow facts within a procedural; and *call*, *return*, and *call-to-return* edges to propagate data-flow facts inter-procedurally.

FlowDroid performs a forward IFDS pass to propagate tainted access paths together with an on-demand backward IFDS pass for discovering aliases, until a fixed point is reached. In Figure 1, a new tainted access path `a` is generated at line 2 ($< 2^{\bullet}, \mathbf{0} > \rightarrow < 2_{\bullet}, \texttt{a} >$) and `o1.g` is tainted by `a` at line 8 ($< 8^{\bullet}, \texttt{a} > \rightarrow < 8_{\bullet}, \texttt{o1.g} >$) as computed by the forward IFDS pass. At line 8, when storing `a` to `o1.g`, FlowDroid starts a backward IFDS pass to search for the aliases of `o1.g`, resulting a new tainted access path `o2.f.g` generated by the statement `o2.f = o1` (line 5). The new access path is then propagated forwardly and recognized as being tainted just after line 8. As a result, both `b` and `c` are tainted at the sink point at line 14 and 15, respectively.

Implementation: FlowDroid is implemented in SOOT and operates on SOOT's Jimple IR. The class `AccessPath` implements access paths. All access paths are abstracted with *k-limiting*, where by default, k is set to 5. In the underlying IFDS solver, the path edge $< s_p, d_1 > \rightarrow < n, d_2 >$ is implemented by an object of class `PathEdge` which contains 3 fields for the source fact d_1, target fact d_2, and the target location n, respectively. Note that the source location s_p can be uniquely identified by n. The set of path edges *PathEdge* is stored in a hash map whose key refers to a path edge and value is its target node. The set of summary edges $\mathcal{S}$ are not explicitly stored and they are handled the same as normal intra-procedural flow edges. *Incoming* is stored in a hash map whose key is a pair $< m, d_1 >$ where m is a method and d_1 is a data-flow fact, representing the target node of a call flow edge. Its value is the set of all predecessors of $< m_p, d_1 >$ implemented as a map, whose key is a predecessor node $< c, d_2 >$ and value is a data-flow fact d_0. The tuple $< d_0, d_2, c >$ can uniquely identify the path edge in $proc(c)$ reaching $< c, d_2 >$. *EndSum* is stored in a similar fashion as *Incoming*.

A path edge object (of class `PathEdge`) is uniquely hashed according to its source fact, target fact, and target location. When propagating the path edge $< s_p, d_1 > \rightarrow < n, d_2 >$, FlowDroid firstly creates object `p` of class `PathEdge`, then the hash code of `p` is computed and looked up in the hash map. If there already exists an entry with the same hash code, `p` is skipped and will be reclaimed by JVM's garbage collector.



```
void main() {
    a = source();
    o1 = new Object();
    o2 = new Object();
    o2.f = o1;
    while() {
        ...
        o1.g = a;
        ...
    }
    b = foo(o1.g);
    ...
    c = foo(o2.f.g);
    sink(b);
    sink(c);
}
```

```
void foo(p) {
    ...
    r = p;
    ...
    return r;
}
```

Fig. 1. FlowDroid — IFDS-based taint analysis

TABLE I
2,053 APPS GROUPED BY FLOWDROID'S MEMORY FOOTPRINT IN ANALYZING EACH APP.

# Total apps		2,053	
Mem	#Apps	Mem	#Apps
NA	825	20G-30G	1
<10G	1,047	30G-60G	5
10G-20G	13	>128G	162

III. Studying Memory Usages

To understand the memory usages of IFDS solvers, we conduct an extensive study by running the latest version of FlowDroid (fa6e25d) [27] on the set of all 2,053 apps from F-Droid [26], an open source Android app repository. The default configuration of FlowDroid (access path length set to 5) is used in analyzing each app. All experiments in this paper are conducted on an Intel Xeon E7-4809v3 (2.0GHz) server with 128GB RAM and 2× 1TB hard-disk drive in Raid1, running on Centos 7. The maximum heap size of JVM is set to 128GB (with `-Xmx`).

Table I summarizes the results, where the 2,053 apps are grouped by memory usages (reported by FlowDroid) of the analysis. After analyzing an app, FlowDroid will report its memory usage, which is the difference between the amount of total memory and that of the currently available memory. For each app, we run FlowDroid 5 times and report the average of the 5 runs. In our experience, the amount of actual physical memory required is always greater than that of the memory usage reported by FlowDroid.

Among the 2,053 apps, 825 apps are not applicable since they either do not require IFDS solvers (i.e., no tainted source or sink) or cannot be processed by SOOT's front-end, 1,047 apps are small and can be analyzed by FlowDroid with less than 10GB of memory (Column 2). However, there are 19 apps taking 10 to 128 GB of memory to analyze and FlowDroid cannot analyze the other 162 apps under the memory budget of 128GB.

We further investigate the 19 apps with memory requirements from 10GB to 128 GB. Table II presents the results. For clarity of presentation, each app is given an abbreviated name (Column 3) and we only use the abbreviated names hereafter. The sizes of those apps (Column 5) range from 348 KB (`CAT`) to 28 MB (`CGAB`), and the memory requirements for analyzing those apps (Column 4) range from 10,823MB (`NMW`) to 44,905MB (`CGT`). The memory footprints and analysis times (Column 8) are closely related to the number of path edges computed. Column 6 and Column 7 present the number of forward and backward path edges, respectively. The app `CGT` has the largest number of total path edges, as well as a long analysis time.

Let us have an in-depth look at how different data structures (i.e., *PathEdge, Incoming, EndSum*) of the IFDS solver con-

TABLE II

STATISTICS OF FLOWDROID IN ANALYZING 19 APPS. ABBR IS THE ABBREVIATED NAME FOR EACH APP, MEM IS THE MEMORY USAGE REPORTED BY FLOWDROID, #FPE AND #BPE ARE THE NUMBER OF FORWARD AND BACKWARD IFDS PATH EDGE COUNTS, RESPECTIVELY.

App	Version	Abbr	Mem (MB)	Size	# FPE	# BPE	Time (s)
bus.chio.wishmaster	1.0.2	BCW	12,110	3.6M	31,855,030	25,279,290	424
com.alfray.timeriffic	1.09.05	CAT	12,441	348K	44,774,904	12,351,293	566
F-Droid	1.1	F-Droid	11,403	7.4M	28,978,612	18,939,414	731
hashengineering.groestlcoin.wallet	7.11.1	HGW	13,897	3.2M	40,763,887	25,447,605	584
nya.miku.wishmaster	1.5.0	NMW	10,823	3.5M	28,897,517	25,137,801	346
org.fdroid.fdroid	1.8-alpha0	OFF	11,392	7.6M	25,725,310	18,388,574	568
org.gateshipone.odyssey	1.1.18	OGO	11,729	2.6M	36,574,830	24,561,384	437
org.lumicall.android	1.13.1	OLA	12,869	5.6M	43,242,840	46,899,396	676
org.yaxim.androidclient	0.9.3	OYA	11,583	1.9M	31,134,795	19,731,055	356
com.github.axet.bookreader	1.12.14	CGAB	19,862	28M	132,406,852	60,651,941	1,655
com.kanedias.vanilla.metadata	1.0.4	CKVM	16,943	6.3M	50,253,185	16,545,672	699
org.secuso.privacyfriendlyweather	2.1.1	OSP	15,654	4.9M	52,555,173	18,637,146	478
org.smssecure.smssecure	0.16.12-unstable	OSS	19,247	14M	67,720,886	62,934,793	2,580
fr.gouv.etalab.mastodon	2.28.1	FGEM	21,669	29M	36,838,257	133,277,513	3,518
com.genonbeta.TrebleShot	1.4.2	CGT	44,905	4.3M	163,539,220	62,170,524	3,212
com.github.axet.callrecorder	1.7.13	CGAC	39,451	5.6M	108,069,294	41,486,114	2,167
com.zeapo.pwdstore	1.3.3	CZP	39,467	4.4M	122,553,741	70,657,317	3,483
de.k3b.android.androFotoFinder	0.8.0.191021	DKAA	41,780	1.5M	95,003,209	88,434,821	3,739
org.kde.kdeconnect_tp	1.13.5	OKKT	32,535	4.5M	38,697,933	25,518,466	811

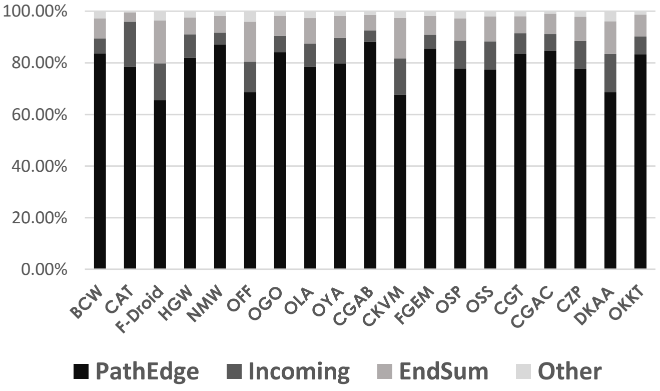


Fig. 2. The memory usage of different data structures, *PathEdge*, *Incoming*, and *EndSum* in the IFDS solver of FlowDroid in analyzing the apps in Table I.

tribute to the total memory usages. We estimate the memory usage of each data structure as the increased available memory after the data structure is freed (by firstly clearing the storage map, then manually invoking `system.gc()` to reclaim objects with no reference). Hence, the memory usage of a data structure is the memory usage of the storage map and those objects only referred to by the map.

Figure 2 shows the memory distribution for each data structure of the IFDS solver. In this experiments, we free the 3 data structures *PathEdge*, *Incoming* and *EndSum* in that order. Path edge objects (of type `PathEdge`) are only referred to by the storage map of *PathEdge* and they are reclaimed after clearing the map. Data-flow fact objects (of type `AccessPath`) can be referred to by any of the 3 data structures. Hence, freeing each structure will result in some data-flow fact objects being reclaimed. As shown in Figure 2, *PathEdge* accounts for the majority of memory usages, consuming averagely 79.07% of the total memory. *Incoming* and *EndSum* use 9.52% and 9.20% of the total memory, respectively.

Observation: PathEdge accounts for the majority of memory consumption in the IFDS solver of FlowDroid. The number of path edges are in the order of hundreds of millions for those apps in Table I, and can reach billions for larger apps. It is not only memory consuming to store the large set of path edges. New path edge objects are frequently created and checked by the IFDS solver. Although hash map lookup has theoretically $O(1)$ time complexity, maps with sizes of hundreds of millions may result in many collisions. Furthermore, it can be time-consuming to frequently compute the hash codes of billions of newly created path edge objects.

IV. REDUCING MEMORY FOOTPRINTS

In this section, we summarize our efforts in reducing the memory footprints of existing IFDS solvers.

Figure 3 overviews our approach, a new taint analysis tool equipped with a disk-assisted IFDS solver. The tool, DiskDroid, employs two memory-saving strategies. Firstly, instead of memorizing all path edges, DiskDroid selectively memorize those edges which tend to be frequently looked up via the *Hot Edge Selector*. The memory footprints are thus reduced by recomputing the other *non-hot* path edges on the fly, with some extra computation cost. Secondly, DiskDroid swaps in-memory data to disk when memory usages reach a predefined threshold. The *Disk Scheduler* implements a set of schemes to efficiently swap data in and out of memory, by considering the following concerns:

- Which data to be swapped out of memory?
- At what granularity to swap data?

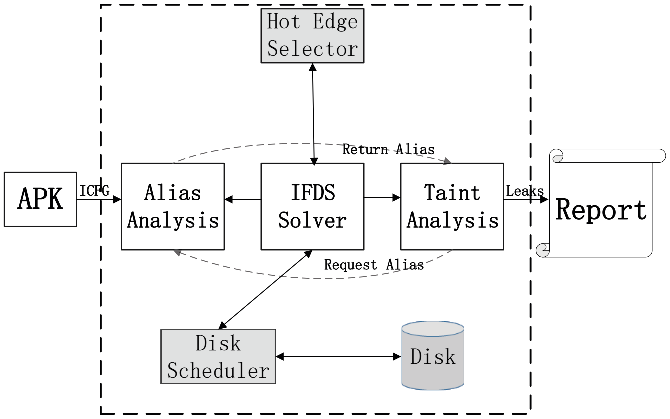


Fig. 3. DiskDroid: taint analysis with disk-assisted IFDS solver.

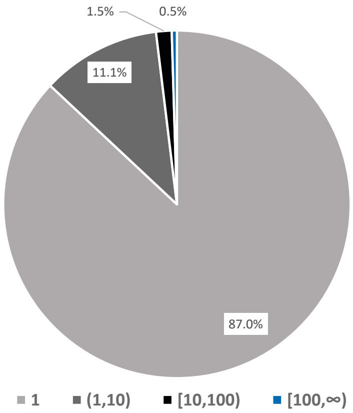


Fig. 4. Distribution of pass edge access number for `CGAB`.

Opportunities and Risks: The key here is to optimize those infrequently accessed or easy-to-compute path edges so that the penalty of recomputation or accessing disks can be minimized. There are many rarely-accessed path edges, as evident in Figure 4. For the benchmark `CGAB`, most of the path edges (86.97%) are visited only once, and only less than 2% of the path edges are visited more than 10 times. However, it is difficult to predicate which path edges are frequently accessed or not. Recomputing or swapping randomly picked path edges may incur a high computational cost, or even result in the algorithm running forever.

A. Hot Edge Selector

The *Hot Edge Selector* makes the classical trade-offs between memoization and recomputation. Algorithm 2 illustrates how to apply this optimization to the traditional Tabulation algorithm (Algorithm 1). The procedure `prop` (lines 11 - 13 in Algorithm 1) is replaced with a new implementation, where non-hot path edges are not memoized (line 12.1). Those non-hot path edges are always inserted into $WorkList$ for further propagation.

We apply the following heuristics to determine whether a path edge $p = <*,*> \rightarrow <n,d>$ is hot or not:

Algorithm 2: IFDS algorithm optimized by hot edge selector.

```
        Procedure Prop($e$):
[12]        if $e$ is not a hot edge :
[12.1]          Insert $e$ into $WorkList$
[12.2]      elif $e \notin PathEdge$ :
[12.3]          Insert $e$ into $WorkList$;Insert $e$ into
                  $PathEdge$
```

1) n is a loop header. Without memorizing path edges leading to loops, the analysis may propagate a path edge within a loop forever and never terminates.
2) p is derived from an inter-procedural flow edge, i.e., n is a function entry, or n is an exit node with d related to the formal parameters of $proc(n)$, or n is a return site with d related to the actual parameters at the callsite. Inter-procedural flows tend to incur a high re-computation cost. For instance, recomputing a path edge at the function entry may result in all other path edges in the callee function being recomputed.
3) p is derived from a backward IFDS pass in FlowDroid. FlowDroid start a backward pass to search for aliases when storing a tainted value to object fields, and aliases identified in the backward pass generate new path edges which are then propagated forwardly. Those new path edges are considered as hot edges to avoid repetitively propagating aliased access paths, which are common in Java programs.

Implementation: It is very efficient to query whether a path edge is hot or not, by examining its target node. In Case 1 and 2, the query returns immediately. In case 3, whenever a new path edge $<*,*> \rightarrow <n,d>$ is generated by a backward IFDS pass, we store the new edge $<n,d>$ in a hash map $\mathcal{D}$, where $d \in \mathcal{D}[n]$. Path edge $<*,*> \rightarrow <n',d'>$ is regarded as a hot edge if $d' \in \mathcal{D}[n']$. Compared to the original algorithm which frequently computes hash codes and looks up *PathEdge* for newly generated path edges (line 12 in Algorithm 1), hot edge queries are much more efficient. As a result, the optimization may result in significant performance speedups (e.g., 58.1% speedup for `CKVM` as shown in Section V).

Theorem 1. *The IFDS algorithm optimized by hot edge selector is sound and can terminate.*

Proof. Soundness: A path edge in the original algorithm is processed at least once in the optimized algorithm. Processing a path edge in both algorithms generates the same set of path edges. As a result, the sets of hot path edges collected by both algorithms are identical.

Termination: When all hot path edges are collected, since path edges at loop/function entries are hot edges, the remaining path edges in $WorkList$ can only propagate along the forward CFG edges. Hence, a path edge p cannot be generated by p' if p' is generated in propagating p. Eventually, the $WorkList$ is $\emptyset$ and the algorithm terminates. □

B. Disk Scheduler

We aim to efficiently swap in-memory data to disk at minimal cost. This can be achieved at two fronts: the first is to store data no longer accessed in disk, and the second is to group *closely-accessed* (i.e., data accessed together within a small interval) data together such that data can be stored to/loaded from disk in a batch. How to identify data with no future accesses and how to discover closely-accessed data? The disk scheduler investigates a variety of grouping and swapping schemes.

1) Grouping: The two data structures $EndSum$ and $Incoming$ are already grouped in the original implementation. Path edges can be grouped according to their containing methods, the source facts, or target facts, as listed below.

- *Method.* Path edges grouped by their containing function, i.e., $\{< s_m, * > \to < *, * >\}$.
- *Method&Source.* Path edges grouped by source nodes, i.e., $\{< s_m, d > \to < *, * >\}$.
- *Method&Target.* Path edges grouped by their containing functions together with the data-flow facts of target nodes, i.e., $\{< s_m, * > \to < *, d >\}$.
- *Source.* Path edges grouped by the data-flow facts of source nodes, i.e., $\{< *, d > \to < *, * >\}$.
- *Target.* Path edges grouped by the data-flow facts of target nodes, i.e., $\{< *, * > \to < *, d >\}$.

In our experience, the grouping strategy *Source* has the best overall performance. The strategy *Method* groups too many path edges together. As a result, it takes each disk access a long time to load the large group of path edges into memory, leading to frequent time outs (in 3 hours). On the other hand, there are only few path edges in *Method&Source* and *Method&Target* groups, resulting in frequent disk accesses and poor performance.

2) Data Swapping: In Algorithm 1, we refer to path edges in $WorkList$ as *active path edges* since they will be further processed. It is necessary to keep active path edges in memory. As shown in Figure 4, most path edges are accessed only once. Hence, it is plausible to swap all inactive edges in $PathEdge$ to disk, since those path edges have already been processed and are unlikely to be accessed again.

DiskDroid swaps path edges in groups, and the notation $g(p)$ denotes the group of path edge p. Memorizing p in $PathEdge$ implies that the whole group $g(p)$ are also in memory. Sometimes it is not sufficient to keep all active groups in memory. Hence, we enforce a *swap ratio* which is the percentage of in-memory groups to be swapped out. After swapping all inactive groups, we select to swap out $g(p)$ where p is at the end of $WorkList$, until reaching the swap ratio (50% by default). Since $WorkList$ is an ordered queue, path edge at the end of the queue are processed last. Thus, its group is swapped out first.

Implementation: To swap path edge in groups, we reorganize the hash map $PathEdge$ into a two-level map. The key of the first-level map is for grouping path edges, and the value is the group of edges, implemented as a hash map in the original form. A path edge group is stored to disk in a separate file, with its name uniquely identified by the group key. Further more, newly created path edge groups (line 12.3, Algorithm 2) are memoized in $NewPathEdge$, in separation from groups loaded from disk ($OldPathEdge$). Such partition enables an efficient way to write path edge groups to disk: groups in $OldPathEdge$ are discarded and a group in $NewPathEdge$ is appended to the file uniquely identified by its key.

A path edge is stored by 3 integer values, one for the source fact, one for the target fact, and one for the target location. We maintain a hash map, together with an array, to get the integer number of a data-flow fact and to restore the data-flow fact from an integer number efficiently. We use the JDK API `BufferedDataInputStream` and `BufferedOutputStream` for reading/writing files.

Disk swapping is triggered when memory usages reach 90% of the given memory budget. To swap out inactive groups, we first traverse $WorkList$ to get all active group keys. Then, the four structures $NewPathEdge$, $OldPathEdge$, $Incoming$, and $EndSum$ are examined. All inactive groups (including path edges groups, and grouped data in $Incoming$ and $EndSum$) are swapped out. When the swap ratio is reached, we invoke `system.gc()` to reclaim memory of objects with no reference.

V. Evaluation

To demonstrate the effectiveness of our disk-assisted IFDS algorithm, we compare DiskDroid against FlowDroid in solving taint analysis problems. The two tools differ in their underlying IFDS solvers only. By Theorem 1, the disk-assisted solver in DiskDroid computes the same data-flow results as the traditional IFDS solver in FlowDroid, and we have validated the correctness of DiskDroid with extensive benchmarking (using DroidBench and open-source Apps). Hence, we will therefore focus on evaluating the performance differences between the two tools.

We apply both tools to the set of 181 apps in F-Droid, with 19 apps requiring 10GB to 128 GB of RAM, and 162 apps requiring more than 128GB of RAM for FlowDroid. In the experiments, we limit the memory usage of DiskDroid to 10GB and that of FlowDroid to 128GB (with `-Xmx`). We run each tool on an app 5 times and report the average data of the 5 runs. Our evaluation answers the following research questions:

RQ1. How is the runtime performance of DiskDroid compared to FlowDroid?

RQ2. Is hot edge optimization effective?

RQ3. Are the grouping and scheduling strategies effective?

A. RQ1. Runtime Performance

Figure 5 compares the run time performance of DiskDroid against FlowDroid for the 19 apps in Table I. Those apps can be analyzed by FlowDroid under the given memory budget of 128 GB. Surprisingly, DiskDroid outperforms FlowDroid by averagely 8.6%, despite given a much smaller memory budget (10GB). The performance differences of the two tools vary

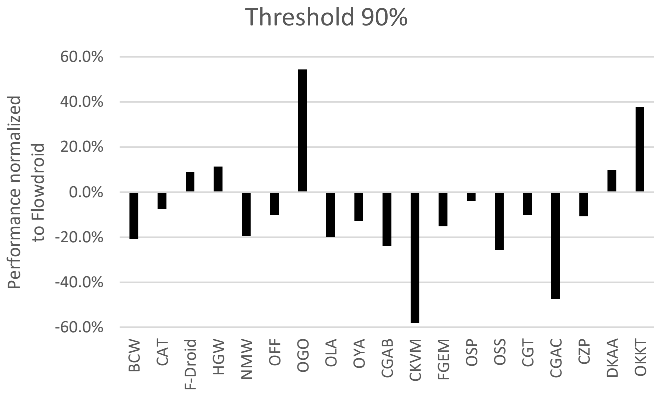


Fig. 5. Performance differences of DiskDroid against FlowDroid. The smaller, the better. Disk swapping is triggered at a usage ration of 90%.

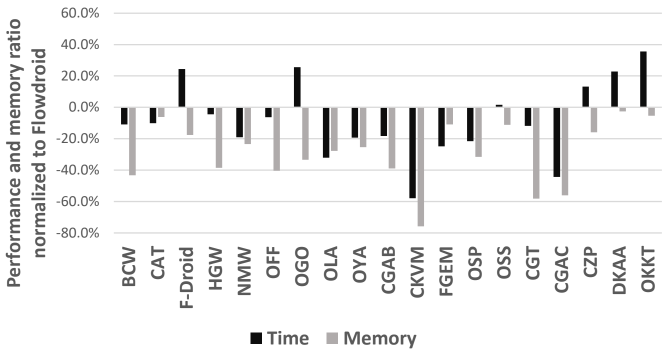


Fig. 6. The differences of run time performance and memory usages of applying hot edge optimization to FlowDroid. The smaller, the better. On average, the optimization saves memory by 30.8%.

TABLE III
NUMBER OF DISK ACCESSES, NUMBER OF PASS EDGE GROUPS AND AVERAGE PATH EDGE GROUP SIZE. #WT AND #RT ARE THE NUMBER OF WRITE AND READ ACCESSES TO DISK, RESPECTIVELY. #PG IS THE NUMBER OF PATH EDGE GROUPS WRITTEN TO DISK AND |PG| IS THE AVERAGE GROUP SIZE.

	# WT	# RT	# PG	|PG|
CAT	2	17,619	194,568	21
F-Droid	2	18,223	492,816	13
HGW	4	33,499	227,523	38
CGAB	2	16,320	120,371	43
CGT	6	51,166	327,411	46
CGAC	4	59,057	321,553	29

widely across the 19 benchmarks, from a large slowdown of 54.5% (OGO) to a significant speedup of 58.1% (CVKM). This is due to the fact that memory-oriented optimizations of DiskDroid are double-edged-swords for runtime performance. For instance, in hot edge optimization, the benefits of skipping hash code computation and hash map lookups for non-hot edges can outweigh the cost in recomputing those edges. Similarly, the cost of swapping data in and out of memory can be offset by the benefits of more efficient hash map lookups after reorganizing the hash map $PathEdge$. Among the 19 apps, there are 13 apps (BCW, CAT, NMW, OFF, OLA, OYA, CGAB, CKVM, FGEM, OSS, CGT, CGAC, CZP) with performance improvements (from 7.4% to 58.1%), 5 apps (F-Droid, HGW, OGO, DKAA, OKKT) with slowdowns (from 9.0% to 54.5%), and 1 app (OSP) with insignificant performance changes.

Table III shows the number of disk accesses and the average group size for 6 apps. Column 2 gives the number of disk-swapping (triggered at memory usage threshold 90%), which is small. There are more frequent read accesses (Column 3): a path edge group is loaded from disk whenever a query fails to locate a path edge in the memoized hash map. However, the ratio is small compared to the number of path edges (Column 7 and Column 8 in Table I). For CGAC with the most number of read accesses (59,057), the ratio is 0.04%, suggesting that most path edge queries are satisfied by look up the memoized hash map. Column 4 shows the number of path edge groups swapped to disk. Compared to Column 3, the number is an order of magnitude larger. This confirms the finding in previous study: most path edges are accessed only once and a majority of path edge groups are stored in disk and never loaded.

We run DiskDroid on the other 162 apps demanding more than 128GB of RAM for FlowDroid. Among the 162 apps, DiskDroid can process 21 of them in 3 hours and the other 141 apps timeout in 3 hours.

B. RQ2. Hot Edge Optimization

We apply hot edge optimization to FlowDroid and report the differences of run time performance and memory usages for the same set of 19 apps. The memory usage is limited to 128GB in both configurations. Figure 6 summarizes the results. The optimization is very effective for the 9 apps CKVM, CGT, CGAC, BCW, OFF, CGAB, HGW, OGO, and OSP, with memory usage reduced from 31.6% (OSP) to 75.8% (CKVM). However, the memory usage improvements are insignificant for the 6 benchmarks CZP, OKKT, OSS, FGEM, CAT, and DKAA, with a memory usage reduction of less than 16%.

Hot edge optimization saves memory at the cost of recomputing non-hot path edges, resulting in an increased number of computations. Table IV compares the number of computations of FlowDroid (Column 2) to that when hot edge optimization is applied (Column 3). The number of computations increases significantly, from 1.08X (CKVM) to 3.33X (CZP). For CKVM, we observe a large performance improvement of 58.1%, suggesting that this optimization has significant positive impacts to run time performance, by reducing the number of hash code computation and hash map lookups. The app CZP has the largest computational increase of 3.33X. However, we observe a speed up of 15.9%. This is because non-hot edges can be computed very efficiently and the extra computational cost is outweighed by the positive impact of the optimization.

C. RQ3. Grouping and Swapping Strategies

Figure 7 compares the run time performance of different grouping schemes. After hot edge optimization, 7 apps (BCW, NMW, OFF, OLA, OYA, OSP, and CKVM) can be analyzed in 10

TABLE IV

NUMBER OF COMPUTED PATH EDGES. #FLOWDROID IS THE ORIGINAL NUMBER OF FLOWDROID, AND #OPTIMIZED IS THE NUMBER WITH HOT EDGE OPTIMIZATION.

	#FlowDroid	#Optimized	Ratio
BCW	32,447,505	44,222,211	1.36
CAT	44,069,465	77,675,474	1.76
F-Droid	29,206,743	38,638,259	1.32
HGW	25,926,773	83,714,388	3.23
NMW	30,813,008	40,804,585	1.32
OFF	25,710,812	34,552,561	1.34
OGO	39,295,629	80,583,394	2.05
OLA	41,666,549	57,461,639	1.38
OYA	29,122,085	32,275,022	1.11
CGAB	132,176,249	275,527,399	2.08
CKVM	38,541,452	41,518,262	1.08
FGEM	37,480,947	85,214,757	2.27
OSP	52,378,247	60,983,905	1.16
OSS	67,487,451	158,045,173	2.34
CGT	160,534,182	517,692,586	3.22
CGAC	103315965	177215471	1.72
CZP	140323650	466792064	3.33
DKAA	93279912	173273865	1.86
OKKT	64216399	131378047	2.05

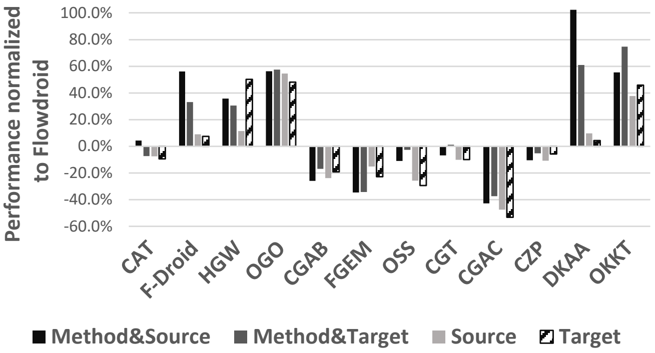


Fig. 7. Performance differences with different grouping schemes. The smaller the better.

GB of RAM. Hence, we report the run times for the other 12 apps in Table I. Among all the grouping schemes, the *Method* scheme performs worst and it frequently timeouts in 3 hours. Figure 7 reports the performance for the other 4 grouping schemes. Since disk scheduler swaps in-memory data to disk at the same threshold for all grouping schemes, the differences of memory usages are negligible thus not reported. Overall, the *Source* scheme exhibits the best run time performance and it is used as the default grouping scheme.

Figure 8 compares the run time performance of different swapping policies with an enforced swap ratio. *Default* is the policy in DiskDroid and *Random* is the policy to randomly swap out path edge groups. The policy *Default 0%* swaps out all inactive groups and keep all active groups in memory. The random policy performs poorly, with a 112% slowdown over FlowDroid for CGT and it timeouts (in 3 hours) for the five apps FGEM, OSS, CZP, DKAA and OKKT. The *Default 0%* policy also fails to analyze these five apps and throws out out-of-memory or gc exceptions. This is because without

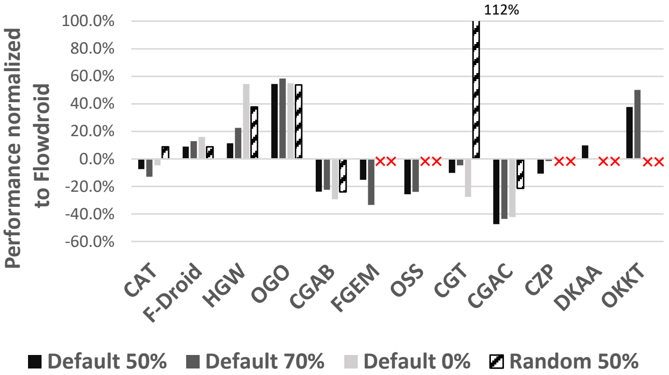


Fig. 8. Performance differences using different swapping policies. Default 50%, Default 70% and Default 0% are the policy in DiskDroid with a enforced swap ratio of 50%, 70%, and 0%, respectively. Random 50% randomly swap out groups with a ratio of 50%.

enforcing a swap ratio, the memory usages easily exceed the threshold, leading to frequent disk swapping and gc operations. Comparing *Default 50%* to *Default 70%*, the run time differences are insignificant when a different swapping ratio (50% vs 70%) is enforced.

Discussion: The memory-oriented optimizations in DiskDroid are effective in terms of reducing memory footprints, enabling the tool to analyze large apps (requiring more than 128 GB of RAM for FlowDroid) with a small memory budget of 10GB of RAM. Those optimizations have both positive and negative impacts on run time performance, resulting in a wide range of performance differences, from a slowdown of 54.5% (OGO), to a speedup of 58.1% (CKVM), with an overall performance improvement of 8.6%. The positive impacts of those optimizations suggest new opportunities in improving analysis efficiency.

VI. RELATED WORK

The IFDS/IDE analysis framework has been applied in a wide range of different applications, including software testing, security analysis, and program verification. Reps et al. [1] initially presented an IFDS analysis framework to solve the inter-procedural, finite, distribute, subset problem. The framework is then generalized in [28] to solve the more general inter-procedural distribute environment problem (IDE), where the data-flow facts are represented by an environment (i.e., a mapping from symbolic to values). Naeem et al. [19] made a few practical extensions to the original algorithm, which is now adopted in popular analysis and compilation frameworks including WALA [2], SOOT [3], and LLVM [4]. Many optimization techniques have been applied to the IFDS algorithm. WALA [2] provides a memory-efficient bit-vector-based solution, Bodden [29] implemented a multi-threaded IFDS/IDE solver in SOOT, and Schubert et al. [30] developed an extendable IFDS/IDE solver for C/C++ programs in LLVM. Dongjie et al. [10] proposed an effective optimization to the IFDS/IDE solver by propagating data-flow facts sparsely, which can drastically improve performance and memory consumption. The disk-assisted approach in this paper can be applied together with those optimization techniques, to further improve memory scalability.

Graspan [21] presents a disk-based approach which leverages graph computing engines [23], [25], [24], [31] in the big data domain for classical pointer and data-flow analyses. The idea is to partition a large graph into small sub-graphs which can be processed in memory. Following this idea, BigSpa [22] performs static analyses in a distributed manner and Grapple [32] applies the approach for path-sensitive analyses. The disk-assisted approach in this paper differs from the above in that we extend existing in-memory analysis algorithms with efficient disk swapping schemes.

Taint analysis aims to detect information flow violations and is implemented in many commercial and open-source security vetting tools. Among the many taint analysis tools for Android [33], [34], [35], [36], [11], [9], FlowDroid remains to be a state-of-the-art taint analysis tool [37]. This paper introduces a new tool, DiskDroid, which extends FlowDroid with a disk-assisted solver. With this extension, apps requiring up to 128GB of RAM can be analyzed within a memory budget of 10GB, making the tool deployable to normal desktop environments.

VII. Conclusion

In this paper, we present a disk-assisted approach to improve memory scalability of existing IFDS algorithms and develop DiskDroid, a new taint analysis tool based on disk-assisted IFDS solver. With two memory-oriented optimizations, DiskDroid can analyze apps requiring up to 128GB of memory for FlowDroid (a state-of-the-art IFDS-based taint analysis tool) under the memory budget of 10GB, with an overall performance improvement of 8.6%. DiskDroid is available at https://github.com/HaofLi/DiskDroid.

In the future, we aim to study disk-assisted approaches to other in-memory analysis algorithms, such as context-sensitive pointer analyses, and symbolic analyses.

Acknowledgment

This work is supported by National Key R&D Program of China (No. 2016YFB1000201), the Foundation for Innovative Research Groups (61521092), and the National Natural Science Foundation of China (61802368, 61872043).

Appendix A
Artifact Appendix

A. Abstract

The artifacts include executables and datasets to conduct all the experiments in the paper titled *Scaling Up the IFDS Algorithm with Efficient Disk-assisted Computing*. We also provide scripts to help reproduce each experiment in the paper.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** Two optimization techniques to the classical IFDS algorithm: hot edge optimization and disk swapping optimization.
- **Program:** Flow-Droid, the popular taint analysis tool, and DiskDroid, a taint analysis tool with our optimized IFDS solver, as well as 18 Android Apps in our evaluation (Figure 5-8, Table 4), are included in the artifacts.
- **Binary:** Jar files of FlowDroid and DiskDroid.
- **Data set:** 18 Android Apps downloaded from F-Droid in our evaluation.
- **Run-time environment:** CentOS 7 with JDK8 and Python3 installed.
- **Hardware:** 128GB RAM And >128GB disk space.
- **Execution:** We provide scripts to run experiments for all benchmarks, as well as scripts to run on a single benchmark.
- **Metrics:** Analysis times and memory usages for FlowDroid and DiskDroid in analyzing each benchmark.
- **Output:** All experimental results in our paper.
- **How much disk space required (approximately)?:** At least 138GB of disk space, including 10GB for out tool, data-sets and result files, as well as 128GB for temporary files.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes to install JDK8, python3, and a few minutes to download our artifacts.
- **How much time is needed to complete experiments (approximately)?:** We provide scripts to run all benchmarks, as well as scripts to run a single benchmark. It takes about 15 days to get results of all benchmarks, and takes from a few hours to 2 days to get results for a single benchmark.
- **Publicly available?:** The executable jar file is publicly available.
- **Code licenses (if publicly available)?:** GPL2.0
- **Data licenses (if publicly available)?:** GPL2.0

C. Description

1) How Delivered: We have packaged all the datasets, pre-built executables, and scripts into DiskDroid-artifacts.tar.gz, which can be downloaded at
https://doi.org/10.6084/m9.figshare.13246316.

2) Hardware Dependencies: The machine should be equipped with at least 128GB RAM and at least 138GB hard-disk.

3) Software Dependencies: It should run on any linux systems, but we recommond CentOS 7.

D. Installation

Before evaluation, JDK8 and Python3 should be installed. We evaluate our tool on **CentOS 7**, so we provide the installation steps for CentOS 7 below. **The following commands(A.4.1, A.4.2) are executed as root user.**

1) Install JDK8:

```
• yum install -y java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

2) Install Python3:

```
• yum install python36 -y
• pip3 install numpy prettytable
```

E. Experiment Workflow

All the datasets, pre-built executables, scripts are packaged in DiskDroid-artifacts.tar.gz at
https://doi.org/10.6084/m9.figshare.13246316.

```
• tar -zxvf DiskDroid-artifacts.tar.gz
• cd diskDroid
```

1) Run All Benchmarks: There are 18 apps in our benchmarks, you can get all experimental results(i.e., data for all figures and tables in our paper) by running the following command.

```
• python3 bin/run.py -t benchmarks/ -k ALL
```

Note that it takes 15 days on our server to finish all experiments. The command will print all experimental data to console.

The Following command can run each experiment and print out the results separately.

```
• python3 bin/run.py -t benchmarks/ -k $
```

$ can be the following items.

- **flowdroid**: Run FlowDroid only, and print out the memory usages, analysis times and number of PathEdges for FlowDroid (Table 2).
- **memoryUsage**: Run FlowDroid only, and summarizes the memory usages of different data structures (Figure 2).
- **pathedgeAccessNum**: Run FlowDroid on CGAB, and report the distribution of pass edge access number (Figure 4).
- **sourceGroup**: Run DiskDroid with default configuration, i.e., group by source (Figure 5, Figure 7, Figure 8), and report the analysis times.
- **onlyHotEdge**: Run DiskDroid with only hot edge optimization enabled (Figure 6, Table 4), and report the analysis times and memory usages.
- **methodSourceGroup**: Run DiskDroid with the Method&Source grouping schemes (Figure 7), and report the analysis times.
- **methodTargetGroup**: Run DiskDroid with the Method&Target grouping schemes (Figure 7), and report the analysis times.
- **targetGroup**: Run DiskDroid with the target grouping schemes (Figure 7), and report the analysis times.
- **Random_50**: Run DiskDroid with the random swapping scheme and an enforced swap ratio of 50% (Figure 8), and report the analysis times.
- **Default_70**: Run DiskDroid with the default swapping scheme and an enforced swap ratio of 70% (Figure 8), and report the analysis times.
- **Default_0**: Run DiskDroid with the default swapping scheme and an enforced swap ratio of 0% (Figure 8), and report the analysis times.

For example, if you want to get Table 2, you can run:

```
• python3 bin/run.py -t benchmarks/ -k flowdroid
```

We remind again that it will take about 3 days to finish one experiment for all the 18 Apps, and can take 15 days to finish all experiments for all the 18 Apps. We also provide a script which runs on a single benchmark, as shown in section A-E2.

2) Run One Benchmark: There are two command line options: `-t` and `-k`. The `-k` option is same as the items which we describe in section A-E1, and the `-t` option is used to specify the target benchmark. For example, to get all experimental results for `com.alfray.timeriffic_10905.apk`, we can run below command.

```
• python3 bin/run-single.py -t
  benchmarks/group1/com.alfray.timeriffic_10905.apk -k
  ALL
```

F. Evaluation and Expected Result

The results are printed to the console. The results are consistently indexed with the paper, please check our discussions in the paper for more details. For figures, we only print out the corresponding data instead of generating graphs.

G. Notes

Because Flowdroid is unstable every run, We run each tool on an app 5 times and report the average data of the 5 runs. So, if evaluating many times, the time and memory may be different.

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*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[2] IBM. Wala: T.j. watson libraries for analysis. http://wala.sourceforge.net. Accessed: 2020.

[3] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011. [Online]. Available: http://www.bodden.de/pubs/lblh11soot.pdf

[4] Llvm framework. https://llvm.org/. Accessed: 2020.

[5] J. Späth, K. Ali, and E. Bodden, "Ideal: Efficient and precise alias-aware dataflow analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133923

[6] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[7] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," ser. ISMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 85–96. [Online]. Available: https://doi.org/10.1145/2491894.2466483

[8] ——, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 343–353. [Online]. Available: https://doi.org/10.1145/2025113.2025160

[9] 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*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[10] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the ifds algorithm with applications to taint analysis," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 267–279. [Online]. Available: https://doi.org/10.1109/ASE.2019.00034

[11] F. Wei, S. Roy, X. Ou, and Robby, "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*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1329–1341. [Online]. Available: https://doi.org/10.1145/2660267.2660357

[12] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," *SIGPLAN Not.*, vol. 44, no. 6, p. 87–97, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542486

[13] T. Tan, Y. Li, Y. Zhang, and J. Xue, "Program Tailoring: Slicing by Sequential Criteria (Artifact)," *Dagstuhl Artifacts Series*, vol. 2, no. 1, pp. 8:1–8:3, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6129

[14] 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.

[15] 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.

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

[17] 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*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 167–177. [Online]. Available: https://doi.org/10.1145/3238147.3238185

[18] A. Gotsman, J. Berdine, and B. Cook, "Interprocedural shape analysis with separated heap abstractions," in *International Static Analysis Symposium*. Springer, 2006, pp. 240–260.

[19] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the ifds algorithm," in *Compiler Construction*, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144.

[20] 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*, ser. ICSE '15. IEEE Press, 2015, p. 426–436.

[21] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "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*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 389–404. [Online]. Available: https://doi.org/10.1145/3037697.3037744

[22] Z. Zuo, R. Gu, X. Jiang, Z. Wang, Y. Huang, L. Wang, and X. Li, "Bigspa: An efficient interprocedural static analysis engine in the cloud," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 771–780.

[23] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 31–46.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 599–613.

[25] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 472–488. [Online]. Available: https://doi.org/10.1145/2517349.2522740

[26] F-droid. https://f-droid.org/. Accessed: 2019.12.

[27] Flowdroid-github. https://github.com/secure-software-engineering/FlowDroid. Accessed: 2019.

[28] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, vol. 167, no. 1, pp. 131 – 170, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397596000722

[29] 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*, ser. SOAP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 3–8. [Online]. Available: https://doi.org/10.1145/2259051.2259052

[30] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[31] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 17–30.

[32] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li, "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303972

[33] 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, no. 201, 2015, p. 110.

[34] 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.

[35] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 280–291.

[36] 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.

[37] 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.