Boosting the Performance of Multi-solver IFDS Algorithms with Flow-Sensitivity Optimizations

Haofeng Li†, Jie Lu†, Haining Meng†‡, Liqing Cao†‡, Lian Li†‡§* and Lin Gao¶
† SKLP, Institute of Computing Technology, CAS, China
‡ University of Chinese Academy of Sciences, China
§ Zhongguancun Laboratory, China
¶ TianqiSoft Inc, China
† {lihaofeng, lujie, menghaining, caoliqing19s, lianli}@ict.ac.cn ¶ gaolin@tianqisoft.cn

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444884

***Abstract*—The IFDS (Inter-procedural, Finite, Distributive, Subset) algorithms are popularly used to solve a wide range of analysis problems. In particular, many interesting problems are formulated as multi-solver IFDS problems which expect multiple interleaved IFDS solvers to work together. For instance, taint analysis requires two IFDS solvers, one forward solver to propagate tainted data-flow facts, and one backward solver to solve alias relations at the same time. For such problems, large amount of additional data-flow facts need to be introduced for flow-sensitivity. This often leads to poor performance and scalability, as evident in our experiments and previous work. In this paper, we propose a novel approach to reduce the number of introduced additional data-flow facts while preserving flow-sensitivity and soundness.**

We have developed a new taint analysis tool, SADROID, and evaluated it on 1,228 open-source Android APPs. Evaluation results show that SADROID significantly outperforms FLOWDROID (the state-of-the-art multi-solver IFDS taint analysis tool) without affecting precision and soundness: the run time performance is sped up by up to 17.89X and memory usage is optimized by up to 9X.

***Index Terms*—multi-solver IFDS, taint analysis, scalability**

I. INTRODUCTION

The IFDS (Interprocedural, Finite, Distributive, Subset) data-flow problem [1] is widely applied in solving a variety range of analysis problems, such as taint analysis [2], [3], [4], [5], bug detection[6], [7], [8], [9], pointer analysis [10], [11], shape analysis [12], and slicing [13]. In IFDS, the set of data-flow facts D is finite and the transferring functions (in $2^D \mapsto 2^D$) distribute over the meet operator $\sqcap$. In practice, IFDS problems are solved flow- and context-sensitively as a special kind of graph-reachability problems (reachable along inter-procedurally realizable paths) [14]. The algorithm has worst-case time complexity of $O(|E||D|^3)$ and space complexity of $O(|E||D|^2)$, where $|E|$ and $|D|$ are the program size and the size of the data-flow facts domain, respectively.

Many analysis problems are formulated as multi-solver IFDS problems. For instance, FLOWDROID [2] formulates taint analysis as a multi-solver IFDS problem, where one IFDS solver propagates tainted data-flow facts (represented as access paths) forwardly, and another backward IFDS solver computes alias relations on-demand. FLOWTWIST [15] employs two IFDS solvers to address integrity and confidentiality issues at the same time. In reality, alias-aware dataflow analyses frequently utilize multiple interleaved IFDS solvers to simultaneously address dataflow equations and aliases. This strategy ensures that aliases are computed precisely and efficiently.

In multi-solver IFDS algorithms, the interleaved IFDS solvers need to be carefully coordinated to ensure flow-sensitivity and context-sensitivity. Context and flow information should be preserved when analysis is handed over from one solver to another. For instance, in FLOWDROID, the forward IFDS solver propagates tainted values forwardly and whenever a tainted value is assigned to heap location `x.f` (such as a field or an array), a backward IFDS solver is spawned to search for aliases until it reaches a heap allocation site. Context-sensitivity is enforced by injecting contexts to the backward solver. When the backward IFDS solver finds an aliased value `y` (via statement `y = x`), a new forward IFDS solver is spawned to propagate the tainted value `y.f` from the location where alias is introduced. For flow-sensitivity, data-flow facts are extended with an *active point* (i.e., the starting program point of the backward IFDS solver) and a value is considered as tainted only after it reaches the active point. When a backward analysis discovers an alias which is then propagated forward, it becomes tainted only when it receives a taint value. It is untainted at the point of alias creation and the alias becomes tainted only at the point from where a backward search for aliases started. Hence an alias is considered tainted (i.e., "is activated") when it reaches an active point. Hence the name active point.

The above extension ensures flow-sensitivity. However, it also introduces a large number of data-flow facts. In our study, the number of data-flow facts can increase up to 9.55X due to active point extensions (the same data-flow fact can be duplicated multiple times at distinct active points). As a result, existing multi-solver IFDS algorithms often suffer from poor performance and scalability. For example, previous study [16] applied FLOWDROID [2] to a set of 2,950 Android apps on a compute server with 730 GB of RAM and 64 Intel Xeon CPU cores, there are still 16 apps unanalyzable because the IFDS solvers take more than 24 hours to finish and consume all the memory for each app. Although recent optimization techniques [3], [17], [18] can significantly improve run-time performance and memory footprints, they still require large

*Corresponding author.

memory budgets (>200 GB) and long analysis times (>3 hours) for many apps.

BOOMERANG [11] performs alias-aware taint analysis by computing aliases (or aliased access paths) on demand – when a tainted value is assigned to heap location `x.f`, it queries all aliased access paths of `x.f` at the assignment statement and propagates them thereafter. In contrast to FLOWDROID, this approach propagates tainted alias values from the initial point where values become tainted, rather than starting from the program point where alias is introduced. Hence, no active point is needed. However, this alternate approach does not guarantee performance improvements because demand-driven flow-sensitive alias analysis is inherently bidirectional [19] and involves multiple iterations of backward and forward IFDS solvers. In fact, BOOMERANG experiences slight slowdown compared to FLOWDROID, as reported in the original paper and confirmed by our experimental results.

This paper presents a new optimization technique which boosts the performance of multi-solver IFDS algorithms. Our approach is simple yet effective, as summarized below.

Observation: the introduction of active point to data-flow facts (e.g., tainted values) can increase the number of data-flow facts by up to 9.55X.
Solution: we apply *flow-sensitivity optimization*, which optimizes performance and memory usages by aggressively removing active points. Thus the number of data-flow facts are significantly reduced. To ensure precision, we search for a realizable path flow-sensitively when taint violation is discovered.

We have developed SADROID, a new taint analysis tool with flow-sensitivity optimization. We evaluated SADROID with a set of 1,228 open source apps. Experimental results show that SADROID can outperform FLOWDROID by 17.89X, and reduce memory footprint by 9X.

This paper makes the following contributions:

- We present a new optimization technique which boosts the performance of multi-solver IFDS algorithms with flow-sensitivity optimization. We unveil an overlooked fact that the flow-sensitive extension in multi-solver IFDS algorithms often introduces a large amount of data-flow facts. Hence, we can achieve significant performance speedups by removing those extensions and precision can be achieved with an efficient post-mortem analysis.
- We implement SADROID, a new taint analysis tool with flow-sensitivity optimizations. We make the tool publicly available on FigShare [1].
- We evaluate SADROID on the set of 1,228 open source apps from F-Droid [20]. Experimental results show that SADROID significantly outperforms FLOWDROID: the analysis time is reduced by up to 17.89X and the memory usage is optimized by up to 9X.

[1]https://doi.org/10.6084/m9.figshare.24654132

The rest of the paper is organized as follows. Section II overviews the classical multi-solver IFDS algorithm in FLOWDROID. Section III highlights how the flow-sensitive active point extension increases the number of data-flow facts and Section IV illustrates our flow-sensitivity optimization. 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 [14] and the multi-solver IFDS algorithm in FLOWDROID [2].

A. The IFDS Algorithm

As in [1], an IFDS instance *IP* is a five-tuple: $IP = (G^*, D, F, M, \sqcap)$, where $G^* = (N^*, E^*)$ is the inter-procedural control flow graph (ICFG) of the program, D is a finite set of data-flow facts, $F \subseteq 2^D \mapsto 2^D$ is a set of functions that are distributive over the meet operator, $M : E^* \mapsto F$ is a map from edges in the ICFG to data-flow functions, and the meet operator $\sqcap$ is either set union or intersection.

The ICFG G^* is built by collecting all the CFGs (control flow graph), G_0, G_1, G_2 ... in 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 methods, and return edges connect exit nodes of callee methods to the *retSite* node. Thus, data-flow facts can propagate inter-procedurally via call and return edges.

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 $E^{\#} = \{\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle \text{ — } m \rightarrow n \in E^*, d_2 \in f(d_1)\}$. Note that here $\mathbf{0}$ 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 $M(m \rightarrow n)$. Dataflow fact d holds at program point n if there exists a realizable path $\langle s_{main}, 0\rangle \rightarrow \langle n, d\rangle$ in $G^{\#}_{IP}$. For efficiency, $G^{\#}_{IP}$ is usually built on demand from G^* during the analysis.

B. The Multi-solver IFDS Algorithm

Algorithm1 reproduces the classical multi-solver IFDS algorithm in FLOWDROID [2]. Similar algorithms are implemented in other analyses with interleaved IFDS solvers [11], [10]. There are two IFDS solvers: the forward IFDS solver (lines 1 – 21) propagates tainted facts and the backward IFDS (lines 22 - 39) solver discovers aliases of tainted facts. Both types of solvers implement the classical Tabulation algorithm which accumulates sets of path edges and summary edges until a fixed point. A path edge is in the form of $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ (lines 3 and 25), where s_p is entry statement of n's procedure p, n is the target statement, and d_1 and d_2 are data-flow facts.

Algorithm 1: The multi-solver IFDS algorithm in FLOWDROID [2].

```
[1]  Algorithm Main loop of forward solver():
[2]      while WorkList_FW ≠ ∅:
[3]          pop ⟨s_p, d_1⟩ → ⟨n, d_2⟩ off WorkList_FW
[4]          switch n:
[5]              case n is call statement:
[6]                  if summary exists for call:
[7]                      apply summary
[8]                  else:
[9]                      map actual parameters to formal
                           parameters
[10]             case n is exit statement:
[11]                 install summary ⟨s_p, d_1⟩ → ⟨n, d_2⟩
[12]                 map formal parameters to actual
                       parameters
[13]                 map return value back to caller's context
[14]             case n is assignment lhs = rhs:
[15]                 d_3 = replace rhs by lhs in d_2
[16]                 if d_3.activepoint is not null:
[17]                     d_4 = d_3
[18]                 else:
[19]                     d_4 = duplicate d_3
[20]                     d_4.activepoint = n
[21]                 insert ⟨s_p, d_1⟩ → ⟨n, d_4⟩ into
                       WorkList_BW
[22]         extend path-edges via the propagate-method of
               the classical IFDS algorithm
[23] Algorithm Main loop of backward solver():
[24]     while WorkList_BW ≠ ∅:
[25]         pop ⟨s_p, d_1⟩ → ⟨n, d_2⟩ off WorkList_BW
[26]         switch n:
[27]             case n is call statement:
[28]                 if summary exists for call:
[29]                     apply summary
[30]                 else:
[31]                     map actual parameters to formal
                           parameters
[32]                 extend path-edges via the
                       propagate-method of the classical IFDS
                       algorithm
[33]             case n is method's first statement:
[34]                 install summary ⟨s_p, d_1⟩ → ⟨n, d_2⟩
[35]                 insert ⟨s_p, d_1⟩ → ⟨n, d_2⟩ into
                       WorkList_FW
[36]                 do not extend path-edges via the
                       propagate-method of the classical IFDS
                       algorithm, killing current taint d_2
[37]             case n is assignment lhs = rhs:
[38]                 d_3 = replace lhs by rhs in d_2
[39]                 insert ⟨s_p, d_1⟩ → ⟨n, d_3⟩ into
                       WorkList_FW
[40]                 extend path-edges via the
                       propagate-method of the classical IFDS
                       algorithm
```

It represents the suffix of a realizable path from $\langle s_{main}, 0\rangle$ to $\langle n, d_2\rangle$.

The forward solver processes each path edge $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ in the worklist $WorkList_{FW}$ one by one. The algorithm takes different actions at different types of program point n: 1) n is a call statement. Function summary will be applied if exists (lines 6 and 7). Otherwise, data-flow facts flow into the callee function (lines 8 and 9) inter-procedurally; 2) n is a function exit. Function summary is constructed (line 11) and data-flow facts flow to the caller function (lines 12 and 13); 3) n is an assignment to heap locations. Note that the data-flow fact d_2 is extended with a field $activepoint$ for flow-sensitivity. If $d_2.activepoint$ is $null$ (line 18), suggesting that d_2 is already activated, the data-flow fact is deeply cloned into a new data-flow fact d_4 with active point n (line 19 and 20). Then a new backward path edge is inserted in $WorkList_{BW}$ to discover aliases (line 21).

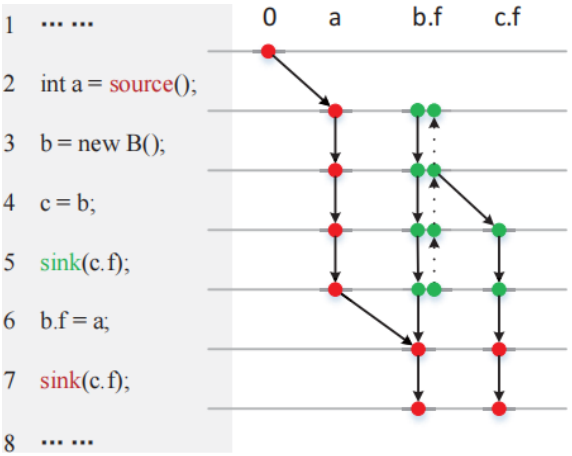

Fig. 1. FLOWDROID — taint analysis as a multi-solver IFDS problem.

Similarly, the backward solver processes path edges in $Work\text{-}List_{BW}$. When it reaches the entry point of a method (line 33) or an assignment, a function summary is constructed and a forward path edge is introduced at the caller (line 35). When it reaches an assignment statement (line 37), a forward path edge is introduced to propagate aliased taint values forward. This way, backward analysis never returns back to callers and context-sensitivity can be easily guaranteed with injected context in the backward solver.

C. Flow-Sensitivity

Figure 1 depicts how flow-sensitivity is achieved when the two IFDS solvers interacting with each other. Each node represents a data-flow fact in the super-graph $G^{\#}_{IP}$ (i.e., a tainted data-flow fact) at a program point and edges propagate facts along the ICFG of the program. The meet operator $\sqcap$ is $\cup$ since a fact 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 6 generates data-flow fact `b.f` by assigning the tainted value `a` to `b.f`), and killed by reset it to an untainted value. Those transfer functions are encoded as edges in $G^{\#}_{IP}$. For instance, the transferring function at line 6 is represented by the edge $\langle 6^{\bullet}, \texttt{a}\rangle \rightarrow \langle 6_{\bullet}, \texttt{b.f}\rangle$, where $6^{\bullet}$ and $6_{\bullet}$ denote the program points before and after the statement at line 6, respectively.

In Figure 1, `a` is tainted by calling the sensitive method `source` (line 2). At line 6, `b.f` is tainted by the assignment statement `b.f = a`. In addition, `b.f` is deeply cloned into a new data-flow fact $\texttt{b.f}_2$ with active point line 6 (green dot) and a new backward path edge is introduced to search

for aliases of `b.f`. The backward IFDS solver searches for aliases until it reaches the heap allocation site at line 2 (`b = new B()`). During the backward analysis, a new aliased value `c.f` is discovered at line 4. As a result, $\langle s_p, \mathbf{0}\rangle \rightarrow \langle 5^{\bullet}, c.f\rangle$ is added into the $WorkList_{FW}$ to propagate the tainted value `c.f` forwardly. The data-flow fact `c.f` is only activated when the forward analysis reaches its active point (line 6). Thus, we can report a taint violation at line 7 since the sensitive data can reach the pre-defined `sink` function. Note that line 5 will not trigger a taint violation since the data-flow fact is not yet activated.

Demand-Driven Alias Analysis: BOOMERANG employs a demand-driven analysis to query aliases at the program point where tainted values are stored to heap. Take Figure 1 for example. Variable `a` is tainted at line 2 and the tainted value is propagated along the ICFG. When `a` is stored to heap location `b.f` at line 6, a demand is raised to compute all aliases (aliased access paths) of `b.f` at line 6, which returns the two access paths `b.f` and `c.f`. Next, both `b.f` and `c.f` are propagated along the ICFG from line 6, and the taint violation at line 7 is discovered.

Since BOOMERANG propagates tainted values from the program point where values become tainted, active point is no longer needed. However, due to the inherent complexity of demand-driven flow-sensitive alias analysis, this approach does not suggest performance improvements. PDFSA [19] has systematically formulates demand-driven flow-sensitive alias analysis as bidirectional analysis, which involves multiple iterations of forward and backward dataflow analyses. For instance, in Figure 1, to query all aliased access paths of `b.f` at line 6, the demand-driven alias analysis firstly invokes a backward IFDS solver to compute aliased values of `b`. As a result, the aliased value `c` is discovered at line 4. Next, a forward IFDS analysis is spawned to propagate alias information along the ICFG. If the forward analysis encounters a heap store instruction of `c`, say `d.f = c`, a new backward IFDS solver will be invoked to compute aliased value of `d`, and so on.

Implementation: FLOWDROID is implemented in SOOT and uses Heros [21], a scalable, multi-threaded implementation of the IFDS framework. The data-flow fact, i.e., tainted values, are implemented by the `Abstraction` class which is a tuple of two elements ⟨`Access-Path, activationUnit`⟩. `AccessPath` represents tainted values as access paths, which are abstracted with *k-limiting* (by default, k is set to 5), and `activationUnit` is the active point which is the source location indicating when a data-flow fact becomes active.

If a data-flow fact is active, field `activationUnit` is null. When starting a backward analysis at program point n, active data-flow facts are deeply cloned with `activationUnit` set to n. Lines 16 - 20 in Algorithm1 highlights the process. In reality, a data-flow fact is often cloned multiple times (up to hundreds of times) at different active points, resulting in a large number of data-flow facts.

TABLE I
2,053 APPS GROUPED BY MEMORY USAGES OF FLOWDROID.

Categories	<8G	8G-32G	>32G	Total
Connectivity	67	0	8	75
Development	34	0	2	36
Games	57	1	4	62
Graphics	15	0	4	19
Internet	218	6	63	287
Money	31	2	11	44
Multimedia	124	3	37	164
Navigation	67	2	14	83
Phone & SMS	36	0	10	46
Reading	61	0	17	78
Science &Education	66	2	10	78
Security	68	0	9	77
Sports & Health	29	1	8	38
System	157	7	24	188
Theming	30	1	3	34
Time	63	4	11	78
Writing	49	1	11	61
Total	987	22	219	1,228

III. MOTIVATION

To understand the performance impacts of active points, we conduct an extensive study by running the latest version of FLOWDROID [22] (a6e25d) on the set of all 2,053 apps downloaded from F-Droid [20], an open source Android app repository. We use the default configuration of FLOWDROID (e.g., access path is limited by 5 and out-of-memory warning is issued when memory usage reaches 90% of the given budget). All experiments in this paper are conducted on an Intel Core (TM) i5-10210U (1.6 GHz) notebook with 40 GB RAM, running on Ubuntu 20.04.1. The maximum heap size of JVM is set to 32 GB (with `-Xmx`).

Table I summarizes the memory usages of FLOWDROID in analyzing the set of 2,053 apps in 17 different categories [20] (Column 1). For each app, it maybe belong to several categories. Following previous work[3], [17], for each app, we run FLOWDROID 5 times and report the average memory usage. Columns 2 to 4 show the number of apps grouped by FLOWDROID's memory usages and Column 5 gives the total number of analyzed apps in each group.

There are 825 apps not processed by FLOWDROID since they either cannot be handled by SOOT or do not contain any sources or sinks. Among the other 1,228 apps, 987 apps can be analyzed within 8 GB of memory (Column 2), 22 apps require a memory footprint from 8 GB to 32 GB (Column 3), and 219 apps cannot be analyzed by FLOWDROID given a memory budget of 32 GB (Column 4). If FLOWDROID runs out of memory for more than 3 times in analyzing an app, we regard the app as not analyzable by FLOWDROID within 32 GB memory budget (i.e., >32 GB).

We further investigate the 22 apps with memory footprints from 8 GB to 32 GB. Table II (Rows 2-23) presents the results. For clarity, we give each app an abbreviated name (Column 3). The sizes of those apk files (Column 5) range from 348 KB (`CAT`) to 23 MB (`CRG`), and the memory requirements for analyzing those apps (Column 4) range from 8.352 MB (`OZT`) to 24.212 MB (`DWAL`).

TABLE II
STATISTICS OF FLOWDROID IN ANALYZING 22 APPS. ABBR IS THE ABBREVIATED NAME FOR EACH APP, MEM IS THE MEMORY USAGE REPORTED BY FLOWDROID, #NF AND #NAP ARE THE NUMBER OF DATA-FLOW FACTS AND THE NUMBER OF ACCESS PATHS, RESPECTIVELY.

App	Version	Abbr	Mem (MB)	Size	# NF	# NAP	NF / NAP	Time (s)
bus.chio.wishmaster	1.0.2	BCW	21,317	3.6M	224,716	34,410	6.53	1,105
ch.hgdev.toposuite	1.2.0	CHT	17,073	2.2M	73,470	10,072	7.29	340
com.alfray.timeriffic	1.09.05	CAT	22,694	348K	180,677	37,479	4.82	782
com.stripe1.xmouse	2.0.27	CSX	11,686	3.5M	128,903	51,573	2.50	318
com.fastebro.androidrgbtool	1.4.4	CFA	15,216	1.8M	174,597	61,257	2.85	978
com.github.quarck.calnotify	5.0.5	CGQC	23,100	2.8M	238,846	75,021	3.06	646
com.liato.bankdroid	1.9.10.6	CLB	11,386	4.1M	651,809	129,943	5.02	300
community.fairphone.launcher	2.0	CFL	8,722	16.0M	189,751	33,174	5.72	153
com.rareventure.gps2	1.1.48	CRG	11,461	23.0M	238,421	77,963	3.06	303
de.k3b.android.locationMapViewer	0.3.5.170911	DKAL	9,385	1.3M	267,154	69,669	3.83	258
de.wikilab.android.ldapsync	2.1.4	DWAL	24,212	1.6M	43,199	11,094	3.89	1,481
eu.pretix.pretixscan.droid	1.2.1	EPPD	10,197	14.0M	301,885	135,778	2.22	140
eu.quelltext.mundraub	v1.230	EQM	12,002	3.0M	70,600	16,644	4.24	533
F-Droid	1.1	FD	16,587	7.4M	443,674	108,404	4.11	656
io.mrarm.irc	0.5.2	IMI	22,482	2.5M	671,410	259,589	2.59	1,732
jwtc.android.chess	8.9.5	JAC	9,032	3.2M	318,144	67,356	4.72	213
net.eneiluj.moneybuster	0.0.15	NEM	13,807	9.0M	73,330	72,834	1.01	448
org.fdroid.fdroid	1.8-alpha0	OFF	14,218	7.6M	292,929	89,209	3.28	606
org.secuso.privacyfriendlyweather	2.1.1	OSP	21,634	4.9M	948,023	108,566	8.73	765
org.secuso.privacyfriendlytodolist	2.2	OSP2	11,466	2.3M	161,547	16,921	9.55	260
org.zephyrsoft.trackworktime	1.0.5	OZT	8,352	2.1M	79,415	32,436	2.45	266
ru.equestriadev.mgke	3.1 Release	REM	15,623	2.4M	132,449	46,879	2.83	430

Impacts of Active Points: To study the impacts of active points, we compare the number of data-flow facts with that number when active points are removed. Specifically, when the multi-solver IFDS analysis reaches a fix point, we count the number of all data-flow facts and the number of all distinct access paths. Columns 6-8 in Table II show the comparison results. The number of data-flow facts (# NF, Column 6) is significantly greater than the number of access paths (# NAP, Column7), up to 9.55X for `OSP2`. There is only slight difference for the benchmark `NEM`, because aliases are rare in this app.

As discussed in Section II-C, data-flow facts are copied into new facts at active points when starting backward analysis. Figure 2 studies the distribution of number of data-flow fact copies. Except for `NEM`, the average number of copies ranges from 1.22 (`EPPD`) to 8.55 (`OSP2`).

Table III shows the maximum number of copies for each app. A data-flow fact can be duplicated for at most 156 times (`CHT`). It may sound surprising why a data-flow fact can be duplicated so many times. Take the data-flow fact `r1.outputStream` in `BCW`, for example. The class of `r1` (`Output`) has two fields `outputStream` and `buffer` (a byte array), `outputStream` write `buffer` to the target file. There are 17 methods in class `Output` with statements storing data into `buffer`, resulting a large number of active points and data-flow fact copies. In addition, the class `Output` is extended by its child class `KryoOutputHC` which introduces new methods and statements storing data into `buffer`.

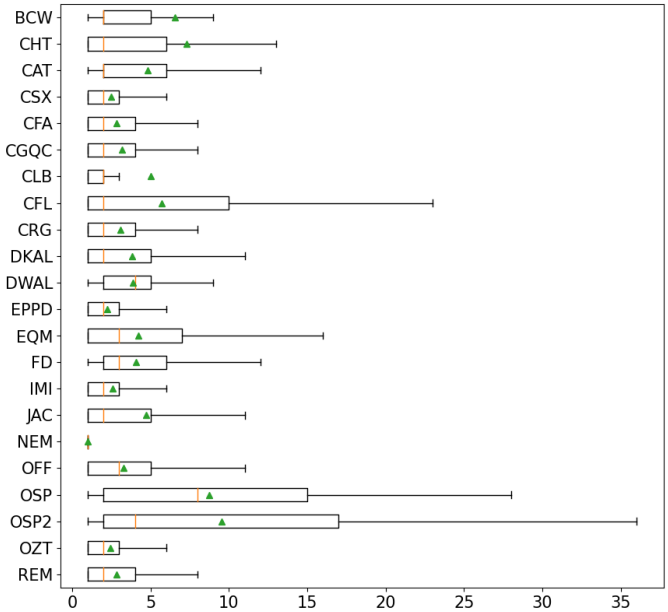

Fig. 2. The number of data-flow fact copies for the `22` apps in Table II.

> Active points, introduced for flow-sensitivity, can have significant impacts to run time performance and memory usages. The number of data-flow facts are inceased by up to 9.55X.

IV. FLOW-SENSITIVITY OPTIMIZATION

Figure 3 overviews SADROID, a new taint analysis tool with flow-sensitivity optimization. SADROID extends the original multi-solver IFDS algorithm with two new modules (high-

TABLE III
THE MAXIMUM NUMBER OF DATA-FLOW FACT COPIES FOR THE 22 APPS IN TABLE II. # MAX IS MAXIMUM NUMBER.

App	BCW	CHT	CAT	CSX	CFA	CGQC	CLB	CFL	CRG	DKAL	DWAL
# MAX	115	156	24	21	13	34	98	45	33	43	26

App	EPPD	EQM	FD	IMI	JAC	NEM	OFF	OSP	OSP2	OZT	REM
# MAX	14	33	24	123	54	7	18	28	36	33	28

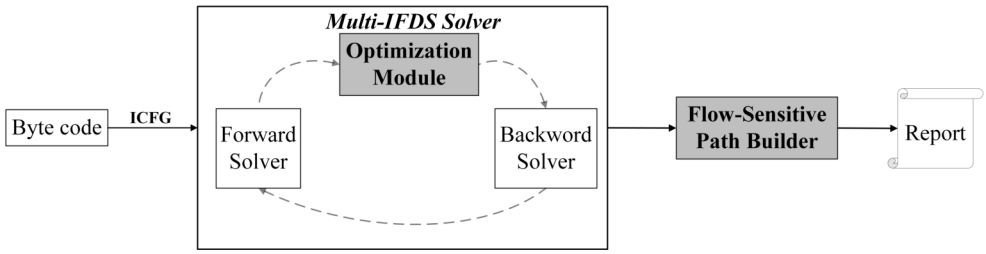

Fig. 3. SADROID: taint analysis with flow-sensitivity optimization.

lighted in gray box): the *Optimization Module* simplifies data-flow fact by disregarding active points (i.e., the field `activationUnit` is always null). This aggressive optimization can largely improve run time performance with a much smaller number of data-flow facts. However, it will also introduce false positives. Hence, *flow-sensitive path builder* will search for a bug-triggering path flow-sensitively when a taint violation is identified. The reported results are guaranteed to be precise.

In practice, the extra overhead introduced by *flow-sensitive path builder* is negligible. Firstly, the existing path search algorithm of FLOWDROID can be extended efficiently to report a bug-triggering path flow-sensitively. Moreover, there are typically only a small number of taint reports in an app and path builder is not frequently invoked.

A. Optimization Module

Algorithm 2: IFDS solver without active point

```
    Algorithm Main loop of forward solver():
      while WorkList_FW ≠ ∅:
[3]     pop ⟨s_p, d_1⟩ → ⟨n, d_2⟩ off WorkList_FW
[4]     switch n:
            ...
[14]        case n is assignment lhs = rhs:
[15]            d_3 = replace rhs by lhs in d_2
[19]            insert ⟨s_p, d_1⟩ → ⟨n, d_3⟩
                into WorkList_BW
        ...
```

The *Optimization Module* simplifies data-flow facts by disregarding active points. The field `activationUnit` is simply set to null, suggesting all data-flow facts are active. As a result, data-flow facts only contain access paths.

Algorithm 2 illustrates how to apply this optimization to the original multi-solver IFDS algorithm (Algorithm 1). In the original algorithm, when a tainted value is assigned to heap (lines 16-20 in Algorithm 1), copies of active data-flow facts are introduced with new active point information. With flow-sensitivity optimization, we do not check active points and do not duplicate data-flow facts. The propagated data-flow facts can then be directly handed over to the backward solver for alias discovery (lines 14 -19 in Algorithm 2).

Figure 4 gives an example demonstrating the effectiveness of this optimization. Variable `a` is tainted at line 2 and passed as an actual parameter to the callee functions of the call statement at line 3. The call statements `b.foo` may invoke multiple targets, e.g., B_1`.foo`, B_2`.foo`, ..., B_n`.foo`. All target methods assign the tainted formal parameter `p` to field `this.f` (line 8). As a result, multiple data-flow fact copies are introduced. Next, the backward analysis will search for aliases of those data-flow fact copies. When the backward analysis leaves the callee targets, each data-flow fact copy will generate a new data-flow fact `b.f` since their active points are distinct. With flow-sensitivity optimization, all copies of `b.f` are merged into one data-flow fact. As such, the number of data-flow facts can be substantially reduced.

B. Flow-Sensitive Path Builder

The *Flow-sensitive Path builder* guarantees flow-sensitivity and precision by searching for a taint propagation path in a flow- and context-sensitive manner. Given a taint report $\langle source, sink \rangle$, FLOWDROID reports a propagation path via a backward depth-first traversal from $sink$ to $source$. Beginning from the fact *d* at *sink*, FLOWDROID traverses all predecessors of *d* until it reaches $source$, where predecessors are those facts creating *d* according to data-flow functions (lines 17 - 25).

For context-sensitivity, a stack S is introduced to record the call stack during the depth-first traversal. The stack S is updated in a standard fashion: callsites are pushed and popped at *retSite* nodes (lines 13 - 14) and *Call* nodes (lines 15 - 16), respectively. When visiting a Call node, its callsite needs to match with the top of stack S to guarantee context-sensitivity.

The procedure `PathBuilderForFact` is extended in Algorithm 3 for flow-sensitivity. Since spurious data flow facts can only originate when switching from a forward IFDS

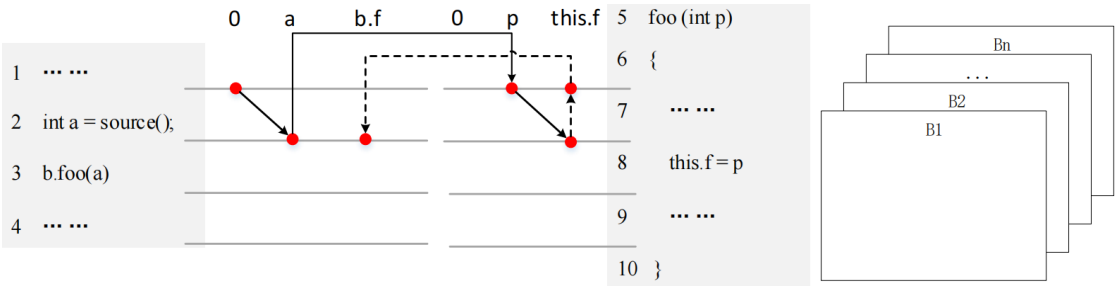

Fig. 4. Flow-sensitivity optimization can reduce a large number of duplicated data-flow facts.

Algorithm 3: Flow-sensitive path builder

```
[1]  Let p be ⟨n_p, d_p⟩, where n_p and d_p are program point
       and data-flow fact for node p, respectively
[2]  Procedure PathBuilder(source, sink):
[3]      path, S := {}
[4]      PathBuilderForFact(sink, path, S)
[5]      return path
[6]  Procedure PathBuilderForFact(d, path, S):
[7]      if d is already visited :
[8]          return FALSE
[9]      if d is source :
[10]         return TRUE
[11]     else:
[12]         append d into path
[13]         if n_d is retSite node at call statement cs :
[14]             push cs to S
[15]         elif n_d is Call node at call statement cs :
[16]             pop cs from S
[17]         for predecessor p of d :
[18]             if n_p is Call node at call statement cs :
[19]                 if cs does not match with S.top :
[20]                     continue
[21]             elif n_p is the fork point of backward solver :
[22]                 if ¬ Valid(p) :
[23]                     continue
[24]             if PathBuilderForFact(p, path, S) :
[25]                 return TRUE
[26]         remove d from path
[27]         if n_d is retSite node at call statement cs :
[28]             pop cs from S
[29]         elif n_d is Call node at call statement cs :
[30]             push cs to S
[31]         return FALSE
[32] Procedure Valid(p):
[33]     Let a taint point be a statement generating d_sink
[34]     Let a kill point be a statement killing d_sink
[35]     Backward traverse the ICFG from n_sink
[36]     if n_sink reaches n_p before a kill point :
[37]         return TRUE
[38]     elif n_sink reaches a taint point before a kill point and
           n_sink can reach n_p :
[39]         return TRUE
[40]     else:
[41]         return FALSE
```

solver to a backward solver, we validate each program point p where a backward solver is spawned (line 22). The validation guarantees that the dataflow fact d_{sink} holds at n_{sink} when continuing the forward analysis from n_p. There are two such cases. In the first case (lines 36 and 37): d_{sink} can flow to n_{sink} along a path from n_p to n_{sink}. Note that in this case d_{sink} already holds at n_p by construction. In the second case (lines 38 and 39), d_{sink} is generated along a path from n_p to n_{sink}. In either case, d_{sink} is guaranteed to hold at n_{sink}.

Theorem 1. *Algorithm 3 guarantees precision and soundness.*

Proof. Precision: By construction, the dataflow fact d_{sink} holds at a spawning point n_p since p is a predecessor of $sink$ in the taint propagation path. The validation function `Valid` further guarantees that d_{sink} can be propagated to n_{sink} from n_p. Thus, no spurious taint propagation path is generated.

Soundness: Let p be a spawning point on a taint propagation path reported by FLOWDROID, the dataflow fact d_{sink} must reaches n_p first then propagated to n_{sink} (i.e., activated by the active point). This condition satisfies the `Valid` procedure and such a path will be identified by Algorithm 3. □

Let us examine the example in Figure 1. After removing active points, there is a taint propagation path $\langle 2, a\rangle \rightarrow \langle 6, b.f\rangle \rightarrow \langle 4, c.f\rangle \rightarrow \langle 5, c.f\rangle$ where $\langle 6, b.f\rangle$ spawns a backward solver. However, the node $\langle 6, b.f\rangle$ is invalid since line 6 cannot reach the sink point (line 5) along the ICFG of the program. Hence, it is an invalid propagation path and we will not report a taint warning at line 5. There is another tainted propagation path in the example $\langle 2, a\rangle \rightarrow \langle 6, b.f\rangle \rightarrow \langle 4, c.f\rangle \rightarrow \langle 7, c.f\rangle$. In this path, the spawning point $\langle 6, b.f\rangle$ is valid since the dataflow fact $c.f.g$ can reach the sink point (line 7) from line 6 without being killed. As a result, we will correctly report the taint warning at line 7.

Figure 5 further illustrates how the flow-sensitive path builder invalidates spurious taint propagation paths. The spurious taint propagation path $\langle 2, a\rangle \rightarrow \langle 6, b.g\rangle \rightarrow \langle 3, c.f.g\rangle \rightarrow \langle 5, c.f.g\rangle$ will be invalidated since the fact $c.f.g$ cannot reach line 5 from line 6 where a backward solver is spawned. Although there is a reachable path from line 6 to line 5, the dataflow fact $c.f.g$ is killed by the assignment at line 7.

C. Implementation

We implement SADROID by applying the two extensions (Algorithm 2 and Algorithm 3) to FLOWDROID. FLOWDROID implements several active points-guided optimizations, as listed below:

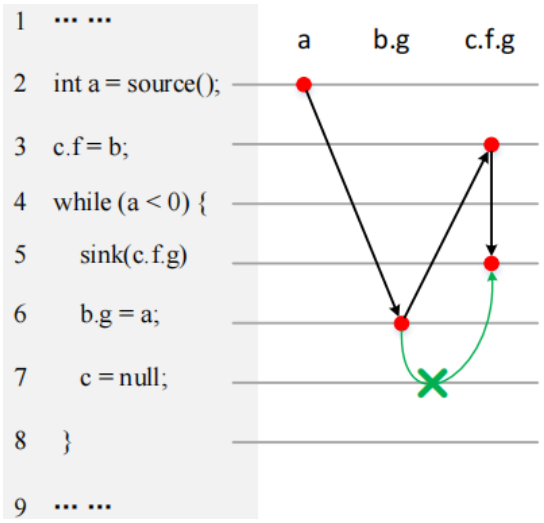

Fig. 5. An example of invalid propagation paths.

1. Inactive facts cannot taint arguments of native call in forward IFDS solver.
2. Inactive facts cannot taint primitive variable in forward IFDS solver.
3. If inactive fact exactly matches with the right hand side value of an assignment, the left hand side value will not be tainted.
4. When backward IFDS solver processes *call* edge, inactive facts with active points in the target method of this function call will not be propagated to the target method.
5. When forward IFDS solver processes *return* edge, inactive facts with active points in current method will not be propagated to the caller method.

The above optimizations are no longer applicable to SADROID. As a result. SADROID will taint more values than FLOWDROID. To address this problem, we further extended path builder as follows:

- For tainted arguments of native calls, we require that all its ancestor nodes can reach the native call without their data-flow facts being killed.
- Exactly matched taint values (with primitive vairables being a special case) are `getField` nodes, and we require that one of its ancestor node is a matched `putField` node which can reach the `getField` node without its data-flow fact being killed.

As such, those tainted values are guaranteed to be active for the first three optimization scenarios.

Note that Optimization 4 and 5 are not sound. The intuition behind the two optimizations is that the active points of those data-flow facts are unlikely to be visited again. Hence, those inactive data-flow facts are discarded. However, in practice, the leaving method may be invoked in a loop or recursive function, and these inactive data-flow facts could be activated later.

V. EVALUATION

To demonstrate the effectiveness of our approach, we compare SADROID against FLOWDROID in solving taint analysis problems. The extensions adopted in SADROID will not impact precision and both tools will produce the same results unless the unsound optimizations in FLOWDROID take effect. We further verified the correctness of SADROID with extensive benchmarking. For DroidBench [23], both FLOWDROID and SADROID report consistent information leaks. Hence, hereafter, we focus on evaluating the performance difference between the two tools.

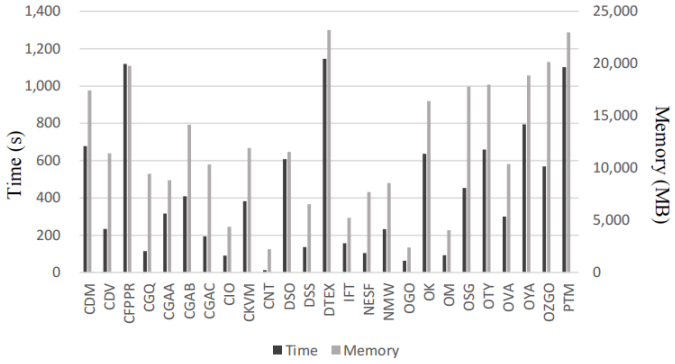

Fig. 6. Run times and memory usages of SADROID for the 25 apps in group 3. The left vertical axis is run time (s) and the right vertical axis is memory usage (MB).

FLOWDROID and its proposed taint analysis are not exclusively designed for running on big servers. They are commonly used as development and testing tools within normal development environments. So, all our experiments are conducted on an Intel Core(TM) i5-10210U laptop (1.6GHz) with 40 GB of RAM, running Unbuntu 20.04.01 with JDK 8. For JVM, we set the maximum heap size to 32 GB (with `-Xmx`). We evaluate each tool using the set of 1,228 apps from F-Droid. Both FLOWDROID and SADROID are implemented based on SOOT, a multi-threaded solver. And both tools exhibit instability with every run. So, for each app, we run both tools 5 times each and report their average run times and memory usages. An app is regarded as unanalyzable by a tool if the tool runs out of memory for more than 3 times. Both tools use the same default configuration file of FLOWDROID to specify taint sources and taint sinks.

Among the 1,228 apps, 194 apps cannot be analyzed by both SADROID and FLOWDROID and those benchmarks will not be discussed further. The remaining 1,034 apps are categorized into 3 groups according to their memory usages of FLOWDROID: group1 consists of 987 apps requiring less than 8 GB of RAM, group2 consists of 22 apps with memory usages from 8 GB to 32 GB, and group3 is a set of 25 apps that can only be analyzed by SADROID within 32 GB of RAM. Our evaluation answers the following research questions:

RQ1. Is SADROID faster than FLOWDROID?
RQ2. Does SADROID consume less memory compared to FLOWDROID?
RQ3. Is flow-sensitive path builder efficient?

A. RQ1. Speedups

Columns 5-7 in Table IV compare the run time of FLOWDROID and SADROID in analyzing the 22 apps in group2.

TABLE IV
PERFORMANCE COMPARISONS BETWEEN FLOWDROID AND SADROID. SAD IS SADROID AND FD IS FLOWDROID.#PE IS THE NUMBER OF PATH EDGES, INCLUDING BOTH FORWARD AND BACKWARD EDGES. # FPB IS THE RUNNING TIME OF FLOW-SENSITIVE PATH BUILDER.

APP	Memory (MB)			Time (s)			# PE			# FPB (s)
	FD	SAD	SAD / FD	FD	SAD	FD / SAD	FD	SAD	FD / SAD	
BCW	21,317	8,871	0.42	1,105	402	2.75	108,679,799	32,540,210	3.34	0.027
CHT	17,073	1,959	0.11	340	19	17.89	72,991,633	4,540,563	16.08	0.012
CAT	22,694	10,245	0.45	782	385	2.03	75,027,983	38,492,671	1.95	0.006
CSX	11,686	11,686	1.00	318	317	1.01	55,425,042	52,310,922	1.06	0.003
CFA	15,216	6,902	0.45	978	158	6.19	45,248,010	10,300,510	4.39	0.010
CGQC	23,100	5,096	0.22	646	150	4.31	114,094,376	26,991,123	4.23	0.052
CLB	11,386	7,094	0.62	300	135	2.22	42,311,595	18,445,617	2.29	0.139
CFL	8,722	2,140	0.25	153	34	4.50	35,328,914	6,482,034	5.45	0.022
CRG	11,461	7,406	0.65	303	130	2.33	33,521,164	16,081,406	2.08	0.009
DKAL	9,385	7,490	0.80	258	74	3.49	49,700,488	15,745,260	3.16	0.007
DWAL	24,212	4,210	0.17	1,481	199	7.44	100,363,660	15,519,292	6.47	0.046
EPPD	10,197	7,544	0.74	140	93	1.51	17,918,040	11,885,048	1.51	0.007
EQM	12,002	11,346	0.94	533	298	1.79	34,565,465	20,140,346	1.72	0.008
FD	16,587	7,085	0.43	656	198	3.31	58,878,170	17,121,861	3.44	0.122
IMI	22,482	9,268	0.41	1,732	828	2.09	88,483,598	44,902,046	1.97	0.002
JAC	9,032	7,454	0.83	213	120	1.78	54,249,091	34,319,652	1.58	0.023
NEM	13,807	OOM	-	448	-	-	50,035,391	-	-	-
OFF	14,218	9,329	0.66	606	193	3.14	53,051,358	15,123,902	3.51	0.122
OSP	21,634	5,814	0.27	765	66	11.59	112,793,448	10,278,503	10.97	0.014
OSP2	11,466	3,853	0.34	260	21	12.38	47,875,686	3,227,889	14.83	0.009
OZT	8,352	4,794	0.57	266	192	1.39	12,526,845	8,462,125	1.48	0.005
REM	15,623	11,406	0.73	430	190	2.26	71,363,571	32,816,220	2.17	0.004
AVG	15,135	7,190	0.48	584	200	2.92	61,161,806	20,748,914	2.95	0.031

Except for the two benchmarks NEM and CSX, SADROID significantly outperforms FLOWDROID, with the largest speedup of 17.89X (CHT). The average speed up is 2.92X. For CSX, the performance difference between SADROID and FLOWDROID is negligible. SADROID runs out of memory on NEM while FLOWDROID can successfully analyze this benchmark. The reason is that the unsound optimizations in FLOWDROID aggressively remove a large number of data-flow facts in analyzing this benchmark.

Columns 8-10 show the number of data-flow facts computed in FLOWDROID and SADROID. Except for CSX and NEM, SADROID can greatly reduce the number of data-flow facts (by averagely 2.95X), resulting in large performance improvements. The number of reduced data-flow facts is small for CSX (1.06X), hence the speedup is only 1.01x.

For the 25 apps (to save space, we omit details of these benchmarks) in group 3, FLOWDROID can not analyze them given a memory budget of 32 GB while SADROID can analyze each app in 20 minutes with the same memory budget. Figure 6 shows the analysis times of SADROID. The average analysis time is 424 seconds and it takes only 13 seconds for SADROID to analyze the benchmark CNT.

Finally, for the 987 apps in group 1, both SADROID and FLOWDROID can analyze them quickly and the performance differences between the two tools are small. For the apps in group1, SADROID takes 9.20 seconds averagely which is less than 14.77 seconds spent by FLOWDROID.

Comparison with Boomerang: We have also compared SADROID against BOOMERANG [24], which extends FLOWDROID with a demand-driven alias analysis. Surprisingly, BOOMERANG cannot run to finish on all benchmarks in group2 and group3. The reason is that BOOMERANG has not been actively maintained and it is not integrated into the latest version of FLOWDROID. Without recent optimizations, earlier versions of FLOWDROID and BOOMERANG cannot scale to large apps in group2 and group3. Hence, we compare the same version of FLOWDROID with BOOMERANG using the first 100 benchmarks in group 1 where benchmarks are sorted by their names alphabetically. BOOMERANG experiences a noticeable slowdown, the average analysis time slows down by 40%. This results are also consistent with the experiments in their original paper [24]. To summarize, demand-driven alias analysis guarantees flow-sensitivity without introducing active points. However, it does not lead to performance improvements.

B. RQ2. Memory Optimization

Columns 2-4 in Table IV and Figure 7 compare the memory usages between FLOWDROID and SADROID for the 22 apps in group2. In this comparison, we use the memory usages reported by FLOWDROID: 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.

As shown in Figure 7, there are 17 apps with significant memory usage reduction, from 26% (OFF) to 89% (CHT). For CHT, we effectively optimize the memory usage by 9X (usage of FLOWDROID divided by usage of SADROID). However, the memory usage improvements are small for the 4 benchmarks CSX, DKAL, EQM, JAC, with a memory usage reduction of

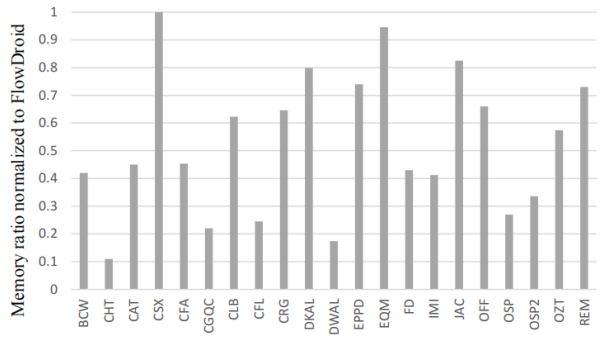

Fig. 7. Memory usage differences between FLOWDROID and SADROID for 21 apps which can be analyzed by both of FLOWDROID and SADROID between 8 and 32 GB of memory.

less than 20%. For the 4 benchmarks, we only reduce a small number data-flow facts (Columns 2-4 in Table IV).

Finally, Figure 6 shows the memory usages for the 25 apps in group3. Those apps cannot be analyzed by FLOWDROID given a memory budget of 32 GB. SADROID can analyze those apps with an average memory consumption of 12 GB. `DTEX` has the largest memory usage of 23 GB, and `CNT` and `DTEX` have the smallest memory usages of 2.24 GB and 2.39 GB, respectively.

C. RQ3. Efficiency of Flow-Sensitive Path Builder

We remove spurious reports introduced by flow-sensitivity optimizations in SADROID by constructing a flow-sensitive propagation path. In practice, the number of taint violations is small and the length of propagation path is not long. Hence, flow-sensitive path builder is very efficient: the overhead is negligible compared to the large amount of data-flow facts optimized.

The last column in Table IV gives the run time of flow-sensitive path builder for the 22 apps in group2. It takes at most 0.139 seconds (`CLB`) to search for flow-sensitive path, and the time is less than 0.1 seconds for all other benchmarks.

Discussion and Limitation: SADROID boosts the performance of multi-solver IFDS algorithm. Compared to FLOWDROID, the performance speedup can be up to 17.89X and memory usage can be optimized by 9X (only 11% of the original memory consumption). Overall, the averagely performance speedup is 2.92X and memory usage is optimized by 2.08X on average. However, SADROID does not guarantee performance improvements. It runs out of memory on one benchmark `NEM` which can be analyzed by FLOWDROID. This outliner is due to the unsound optimizations in FLOWDROID (Section IV-C). To verify it, we run FLOWDROID on `NEM` with those unsound optimizations disabled, and FLOWDROID also runs out of memory.

Although the flow-sensitivity optimization achieves significant performance improvements, there are still 194 apps cannot be analyzed with 32 GB of memory. Other optimization techniques, such as sparsification [3], [18] can be combined with our optimization, to further improve performances.

VI. RELATED WORK

IFDS: The IFDS/IDE analysis framework defines a special data-flow analysis and 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 interprocedural, finite, distribute, subset problem. The framework is then generalized in [25] to solve the more general interprocedural 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. [14] made a few practical extensions to the original algorithm, which is now adopted in popular analysis and compilation frameworks including WALA [26], SOOT [27], and LLVM [28].

Many optimization techniques have been applied to the IFDS algorithm. WALA [26] provides a memory-efficient bit-vector-based solution, Bodden et al. [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. There are a number of research on performance improvement optimizations for IFDS. Dongjie et al. [3] proposed an effective optimization to the IFDS/IDE solver by propagating data-flow facts sparsely, which can drastically improve performance and memory consumption. Arzt et al. [31] pre-computed summaries for library functions, which are applied to improve scalability. Haofeng et al. [17] scaled up IFDS algorithm by removing path edges which may not be accessed anymore and swapping in-memory data to disk when memory usages reaching a pre-defined threshold. Arzt et al. [18] improve the performance of IFDS by removing path edges that are unlikely to be visited in the future.

Multi-solver IFDS: Many analysis problems are formulated as a multi-solver IFDS problems. FLOWDROID [2] (a state-of-the-art multi-IFDS-based taint analysis tool) formulates taint-analysis as a multi-solver IFDS problem: a forward IFDS solver to propagate tainted variables, and a backward IFDS solver to discover aliases. This way, it is context-, flow-, and field-sensitive. Flowtwist [15] employs two IFDS solvers to compute integrity and confidentiality violations at the same time. PDFSA [19] nicely formalizes demand-driven alias problem as bidirectional dataflow problems, suggesting the necessity of multi-solver IFDS for alias-aware data-flow problems.

Data-Flow Facts: Access-paths are effective in track aliases and are widely used to compute alias-aware data flow problems [2], [10]. For performance and scalability, access-paths are often abstracted with k-limiting. Deutsch et al. [32] used regular expressions to symbolically represent an access path, which is effective for recursive data structures (access path `a.f.f.f.f.f` can be simplified as $\texttt{a.f}^5$. Khedker et al. [33] proposed access graphs which can represent infinite lengths of recursive field accesses. Johannes et al. [34] extended the regular expression representation of access path with a set of exclusions for excluded fields. This access-path abstraction can reduce overall number of data-flow facts and

improve summary reuse in IFDS.

In this paper, we present a new optimization techniques for multi-solver IFDS problems. Our optimization reduces the number of data-flow facts by firstly sacrificing flow-sensitivity during the IFDS analysis, then restoring flow-sensitivity by refining the results.

Taint Analysis Tools: 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 [35], [36], [37], [38], [4], [2], FLOWDROID remains to be the state-of-the-art taint analysis tool [39]. This paper introduces a new tool, SADROID, which extends FLOWDROID with flow-sensitivity optimizations to significantly improve performance while preserving precision.

VII. CONCLUSION

In this paper, we propose a simple yet effective optimization technique for multi-solver IFDS algorithms. The optimization is based on the observation that a large number of data-flow facts are duplicated for flow-sensitivity. Hence, our optimization avoids such duplication by sacrificing flow-sensitivity during the multi-solver IFDS analysis and employs a post-mortem analysis to regain flow-sensitivity.

ACKNOWLEDGMENT

We thank all anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (NSFC) under grant number 62132020 and 62202452.

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] 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

[3] 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

[4] 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

[5] 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

[6] 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.

[7] 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.

[8] 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.

[9] 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

[10] 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

[11] 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.

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

[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] 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.

[15] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "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*, 2014, pp. 98–108.

[16] 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.

[17] H. Li, H. Meng, H. Zheng, L. Cao, J. Lu, L. Li, and L. Gao, "Scaling up the ifds algorithm with efficient disk-assisted computing," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 236–247.

[18] S. Arzt, "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)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 1098–1110. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00102

[19] S. Jaiswal, U. P. Khedker, and S. Chakraborty, "Bidirectionality in flow-sensitive demand-driven analysis," *Sci. Comput. Program.*, vol. 190, no. C, may 2020. [Online]. Available: https://doi.org/10.1016/j.scico.2020.102391

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

[21] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012, pp. 3–8.

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

[23] Droidbench 2.0. https://github.com/secure-software-engineering/DroidBench. Accessed: 2017.05.

[24] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java (Artifact)," *Dagstuhl Artifacts Series*, vol. 2, no. 1, pp. 12:1–12:2, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6133

[25] 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

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

[27] 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

[28] Llvm framework. https://llvm.org/. Accessed: 2020.
[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] S. Arzt and E. Bodden, "Stubdroid: automatic inference of precise data-flow summaries for the android framework," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 725–735.
[32] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," *ACM Sigplan Notices*, vol. 29, no. 6, pp. 230–241, 1994.
[33] U. P. Khedker, A. Sanyal, and A. Karkare, "Heap reference analysis using access graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, pp. 1–es, 2007.
[34] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths," in *IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, Nov. 2015, pp. 619–629. [Online]. Available: https://www.bodden.de/pubs/lsb+15access-path.pdf
[35] 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.
[36] 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.
[37] 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.
[38] 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.
[39] 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.