Merge-Replay: Efficient IFDS-Based Taint Analysis by Consolidating Equivalent Value Flows

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

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

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

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00027

***Abstract*—The IFDS-based taint analysis employs two mutually iterative passes: a forward pass that identifies taints and a backward pass that detects aliases. This approach ensures both flow and context sensitivity, leading to remarkable precision. To preserve flow sensitivity, the IFDS-based taint analysis enhances data abstractions with activation statements that pinpoint the moment they acquire taint. Nonetheless, this mechanism can inadvertently introduce equivalent, yet redundant, value flows. This occurs when distinct activation statements are linked with the same data abstraction, resulting in unnecessary computational and memory-intensive demands on the analysis process.**

We introduce MERGEDROID, a novel approach to improve the efficiency of IFDS-based taint analysis by consolidating equivalent value flows. This involves merging activation statements linked to the same data abstraction from various reachable data facts that are reachable at a given program point during the backward pass. This process generates a representative symbolic activation statement applicable to all equivalent data facts, reducing them to a single symbolic data fact. During the forward pass, when this symbolic data fact returns to its point of creation, the analysis reverts to the original data facts alongside their initial activation statements. This merge-and-replay strategy eliminates redundant value flow propagation, resulting in performance gains. Furthermore, we also improve analysis efficiency and precision by leveraging context-sensitive insights from activation statements. Our evaluation on 40 Android apps demonstrates that MERGEDROID significantly enhances IFDS-based taint analysis performance. On average, MERGEDROID accelerates analysis by 9.0× while effectively handling 6 more apps scalably. Additionally, it reduces false positives by significantly decreasing reported leak warnings, achieving an average reduction of 19.2%.

***Index Terms*—Taint analysis, IFDS, scalability, precision.**

I. INTRODUCTION

Static taint analysis is a foundational approach for monitoring the potential flow of sensitive data from program sources to untrusted sinks. It has wide-ranging applications, including memory leak detection [1], SQL injection identification [2], [3], and data leak validation [4], [5] across industries. The analysis algorithm's precision and efficiency significantly impact these applications' effectiveness.

While FLOWDROID [4] stands as a leading tool for static taint analysis of Android apps, its approach can pose computational and memory challenges. It relies on an IFDS-based algorithm [6] with two mutually iterative passes: a forward pass to detect taints and a backward pass to uncover aliases. Each pass is handled by a separate IFDS solver. The forward solver, upon detecting a new taint and needing related aliased taints, invokes the backward solver to search in reverse. Discovered aliased taints are then relayed to the forward solver for further propagation. Although FLOWDROID is known for its high precision due to the flow-sensitive and context-sensitive nature of the IFDS algorithm [6] employed, a previous study [7] revealed that it may struggle to analyze some Android apps, even on a server with 730GB RAM, due to exceeding a 24-hour time budget or memory limitations.

We introduce MERGEDROID, an innovative approach to enhance IFDS-based taint analysis, particularly FLOWDROID. Our focus is on consolidating equivalent value flows, leveraging an unexplored insight that sets it apart from existing FLOWDROID acceleration methods [8]–[11]. In the context of FLOWDROID, the forward solver propagates an aliased taint forward immediately upon discovery by the backward solver. However, it activates the taint only upon reaching the program point where the corresponding alias query was initiated, maintaining flow sensitivity. To achieve this, FLOWDROID employs activation statements to decorate data abstractions (i.e., statements triggering alias queries) and discern their taint activation moments. Nonetheless, as elaborated in Section II, the presence of multiple activation statements linked to the same data abstraction can lead to equivalent and thus redundant value flows. This redundancy unnecessarily heightens the computational and memory demands of the analysis.

Built upon this insight, MERGEDROID employs a simple yet effective *merge-and-replay* algorithm to consolidate equivalent value flows while maintaining precision. This approach is conceptually straightforward. During the backward pass, activation statements linked to the same data abstraction from diverse data facts reachable at a given program point are merged. This yields a representative symbolic activation statement applicable to all equivalent data facts, reducing them to a single symbolic data fact. During the forward pass, as the symbolic data fact propagates back to the same program point, the analysis reverts to the original data facts alongside their initial activation statements. This *merge-and-replay* process eliminates equivalent and redundant value flows originating from activation statements between these two points, all without compromising precision.

∗, † The first two authors contributed equally and are listed in alphabetical order by their last names, while the last two authors share corresponding authorship.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00027

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00027

Establishing suitable merge-and-replay points poses a challenge, particularly given the distributive nature of IFDS-based taint analysis [6]. Deploying merge-and-replay at every program point for all data facts can result in excessive overhead. To tackle this, we opt to strategically implement merge-and-replay solely at callsites. When the backward solver propagates a new data fact from a callee method m' to a callsite c within method m, our *merge* procedure activates. This procedure generates a symbolic activation statement s and produces a corresponding symbolic data fact, after which backward propagation resumes as usual. Subsequently, as the forward solver propagates a symbolic data fact with s as its activation statement to c, just prior to advancing into m', our *replay* procedure comes into play. This contextualizes s in the context of both m and m', executing merge-and-replay in a context-sensitive manner to prevent spurious value flows into callee methods. This results in improved efficiency and precision.

To demonstrate the improved performance of MERGEDROID compared to FLOWDROID, we have evaluated both tools on a set of 40 Android apps, comprising 20 apps from [9], 13 apps from [10], and 7 apps from F-Droid [12]. Each app is allocated a time budget of 3 hours and a memory budget of 256GB. Our results show that MERGEDROID has successfully analyzed 6 apps that FLOWDROID failed to complete due to exceeding the time budget, i.e., OoT (out of time). For the remaining 34 apps, MERGEDROID has significantly reduced the number of path edges and the amount of memory used by an average of 9.7× and 5.2×, respectively. This reduction led to an average speedup of 9.0× for MERGEDROID over FLOWDROID (with the best speedup of 137.9× achieved in one app). Furthermore, MERGEDROID has effectively reduced false positives by significantly decreasing the number of reported leak warnings by an average of 19.2%.

The paper makes the following main contributions:

- A novel *merge-and-replay* algorithm for consolidating equivalent value flows and consequently improving the efficiency and precision of FLOWDROID;
- an open-source implementation of MERGEDROID; and
- an extensive evaluation (in terms of efficiency and precision) of MERGEDROID against FLOWDROID on both micro-benchmarks and real-world Android apps.

II. MOTIVATION

We utilize a carefully constructed example (Section II-A) to illustrate the functioning of FLOWDROID in detecting data leaks (Section II-B), and to demonstrate how MERGEDROID's *merge-and-replay* mechanism operates while highlighting its efficiency and precision advantages (Section II-C).

A. The Motivating Example

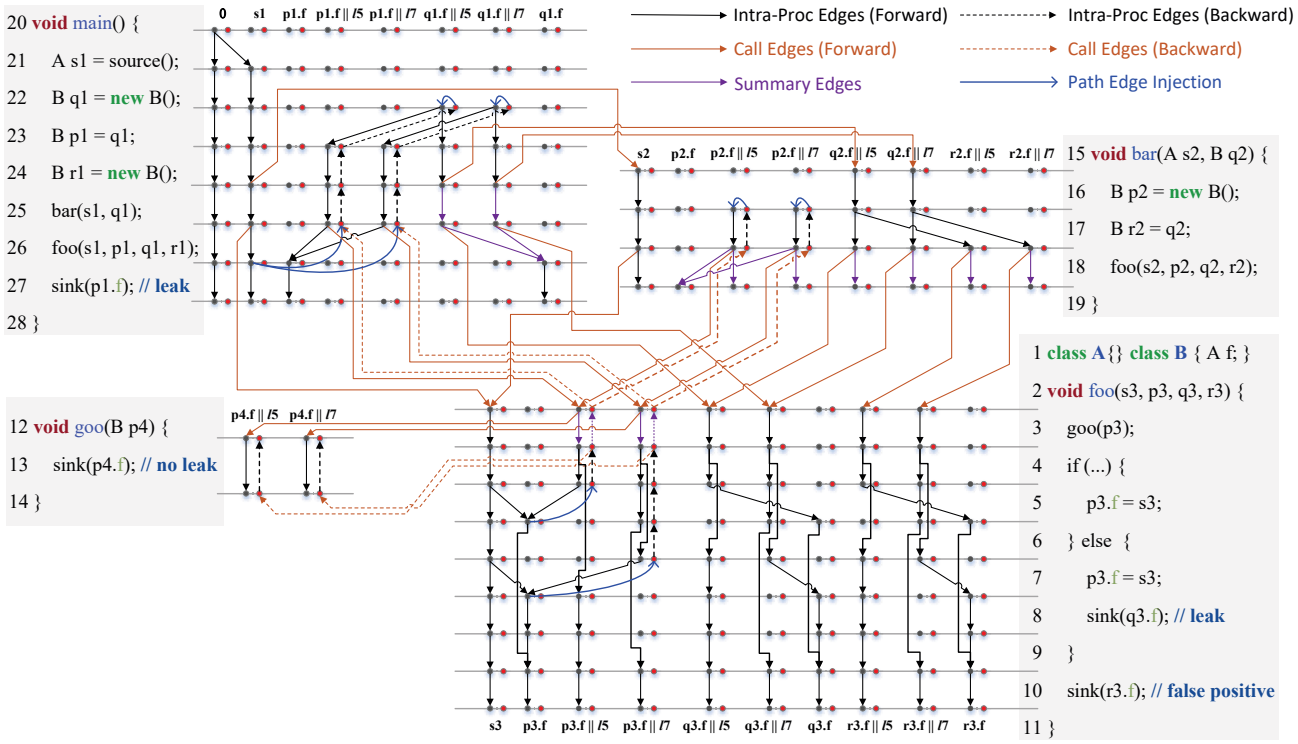
The motivating example, shown in the grey-shaded background of Figure 1, contains two classes, `A` and `B` (line 1), and four static methods: `foo()` (lines 2-11), `goo()` (lines 12-14), `bar()` (lines 15-19), and `main()` (lines 20-28). In `main()`, `s1` is tainted by `source()` and then passed into `bar()` (line 25) and `foo()` (line 26), resulting in `s2` and `s3` being tainted, respectively. It is worth mentioning that `foo()` is invoked twice, first at line 18 and then again at line 26. In `foo()`, `s3` causes `p3.f` to be tainted after lines 5 and 7 in the control flow, leading to a data leak at line 27 (due to `p1` being aliased with `p3`). Meanwhile, a data leak arises at line 8 due to `p1 = q1` at line 23, resulting in `p3` and `q3` being aliased at line 2. Since `goo()` is called before lines 5 and 7, no data leak emerges at line 13. Additionally, no data leaks occur at line 10 due to two reasons: (1) `r3` does not alias any of the other three parameters of `foo()` during its call at line 26, and (2) although `r3` aliases with `q3`, neither `r3` nor `q3` aliases with `p3` during the call to `foo()` at line 18.

For simplicity, lines 5 and 7 are identical. We could have added `s5 = s3` just before line 5 and replaced its `s3` by `s5`.

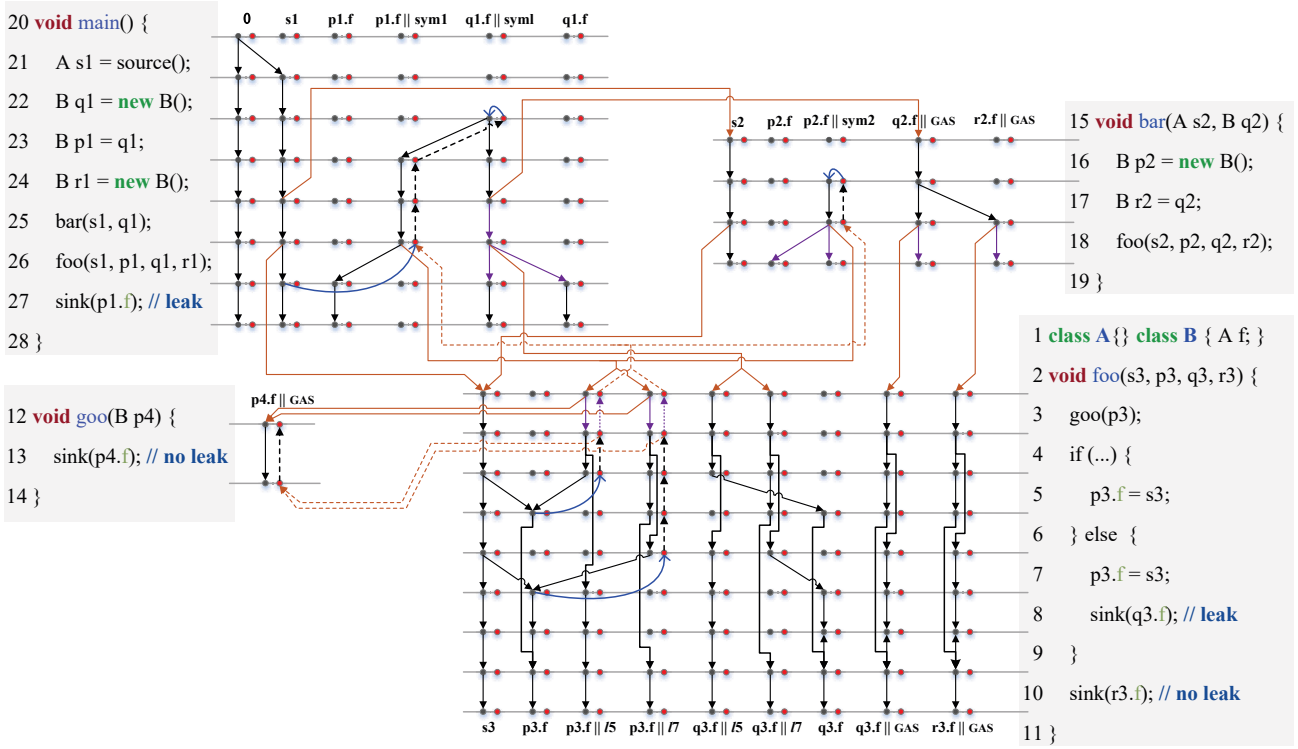
B. FLOWDROID: The IFDS-Based Taint Analysis

To address the taint analysis problem, FLOWDROID [4] employs the conventional IFDS algorithm [6]. The process is depicted in Figure 1a, where a forward pass identifies tainted access paths and a backward pass discovers their aliases. Both passes are simplified into graph reachability tasks on their respective exploded supergraphs, with gray and red nodes representing data facts established in the forward and backward passes, respectively. The two passes operate iteratively, injecting path edges to each other (depicted as →). Note that understanding our example does not necessitate familiarity with the concept of path edges (explained in Section III-A). For clarity, return edges are omitted, and only call edges (→) and summary edges (→) maintain the reachability relationship.

In Figure 1a, only **0** in `main()` is initially reachable. As the forward pass runs, tainted access paths are gradually discovered: `s1` is first tainted by `source()` (line 21), followed by `s2` and `s3` (due to parameter passing at lines 25 and 26), and finally `p3.f` (due to store statements at lines 5 and 7). When FLOWDROID injects a path edge to the backward pass to find aliases of `p3.f`, it attaches an activation statement at line 5 (7) to decorate the data abstraction in order to maintain flow sensitivity, forming a new data fact: `p3.f` $\|\, l5$ (`p3.f` $\|\, l7$). During the backward pass, FLOWDROID identifies the aliases `p4.f` $\|\, l5$, `p2.f` $\|\, l5$, `q1.f` $\|\, l5$, and `p1.f` $\|\, l5$ (`p4.f` $\|\, l7$, `p2.f` $\|\, l7$, `q1.f` $\|\, l7$, and `p1.f` $\|\, l7$) corresponding to `p3.f` $\|\, l5$ (`p3.f` $\|\, l7$). Subsequently, FLOWDROID inserts appropriate path edges at lines 12, 16, 22, and 23—where these aliases are defined—into the forward pass. This integration facilitates the identification of additional tainted access paths: `q1.f` $\|\, l5$, `q1.f` $\|\, l7$, `p1.f` $\|\, l5$, `p1.f` $\|\, l7$, `p1.f`, `q1.f`, `q2.f` $\|\, l5$, `q2.f` $\|\, l7$, `r2.f` $\|\, l5$, `r2.f` $\|\, l7$, `q3.f` $\|\, l5$, `q3.f` $\|\, l7$, `q3.f`, `r3.f` $\|\, l5$, `r3.f` $\|\, l7$, and `r3.f`. Note that `q3.f` $\|\, l5$ and `q3.f` $\|\, l7$ (`r3.f` $\|\, l5$ and `r3.f` $\|\, l7$) become `q3.f` (`r3.f`) after passing through their corresponding activation statements. The callsite at line 26 is also an activation statement for `p1.f` $\|\, l5$, `p1.f` $\|\, l7$, `q1.f` $\|\, l5$, and `q1.f` $\|\, l7$, enabling the conclusion that `p1.f` (`q1.f`) is reachable at the program point before line 27. Based on this taint analysis, FLOWDROID reports three leaks at lines 8, 10, and 27, of which the one at line 10 is a false positive.



(a) FLOWDROID: the classic IFDS-based taint analysis.



(b) MERGEDROID: the IFDS-based taint analysis incorporated with a *merge-and-replay* mechanism.

Fig. 1: Comparing FLOWDROID and MERGEDROID in discovering taints (aliases) in a forward (backward) pass performed on an exploded supergraph with its nodes colored in gray (red). All return edges are omitted to avoid cluttering. Both passes run mutually iteratively via path edge injection. The data abstractions are expressed as k-limited access paths. Active and inactive data facts are represented by da and $da \,||\, as$, where da is a data abstraction and as is an activation statement.

Let us highlight the two limitations of FLOWDROID below:

- **Redundant Propagation of Equivalent Value Flows.** Propagating multiple data facts that share the same data abstraction but have different activation statements, such as $\texttt{p}i\texttt{.f} \,\|\, l5$ and $\texttt{p}i\texttt{.f} \,\|\, l7$, $\texttt{q}j\texttt{.f} \,\|\, l5$ and $\texttt{q}j\texttt{.f} \,\|\, l7$, and $\texttt{r}k\texttt{.f} \,\|\, l5$ and $\texttt{r}k\texttt{.f} \,\|\, l7$, where $1 \leq i \leq 4$, $1 \leq j \leq 3$, and $2 \leq k \leq 3$, is unnecessary and inefficient, as it serves only to waste time and memory resources.
- **Activation Statements without Contexts.** Activation statements in FLOWDROID pinpoint where data facts should be activated, but overlooking their contexts can lead to false positives. When `foo()` is analyzed, both $l5$ and $l7$ serve as activation statements for $\texttt{q1.f} \,\|\, l5$ and $\texttt{q1.f} \,\|\, l7$ during its calls at lines 18 and 26, respectively. However, if `foo()` is invoked at line 26, $\texttt{q1.f} \,\|\, l5$ and $\texttt{q1.f} \,\|\, l7$, initially established during the analysis, will propagate backward to find aliases of `q1`. This causes them to reach the callsite at line 26, progress to line 23 where `p1` and `q1` are identified as aliases, and finally reach `bar()` into which both $\texttt{q1.f} \,\|\, l5$ and $\texttt{q1.f} \,\|\, l7$ flow. This sequence creates a context mismatch, prompting unnecessary propagations and ultimately causing a false positive at line 10 when `foo()` is analyzed again.

C. MERGEDROID *in a Nutshell*

We have designed MERGEDROID that applies a novel *merge-and-replay* algorithm to overcome the two limitations of FLOWDROID, resulting in better efficiency and precision.

In Figure 1b, our *merge-and-replay* approach is illustrated, where it introduces a symbolic activation statement to represent the concrete activation statements of data facts with the same data abstraction propagated from a callee to a caller during the backward pass (**Merge**). For instance, `sym1` and `sym2` are introduced as representative symbolic activation statements when $\texttt{p3.f} \,\|\, l5$ and $\texttt{p3.f} \,\|\, l7$ are propagated from `foo()` into `main()` and `bar()`, respectively. During the forward pass, MERGEDROID switches to the original activation statements of the data facts to continue propagation in callees (**Replay**). For example, when $\texttt{p1.f} \,\|\, \texttt{sym1}$, $\texttt{q1.f} \,\|\, \texttt{sym1}$, and $\texttt{p2.f} \,\|\, \texttt{sym2}$ are propagated back into `foo()`, MERGEDROID recovers the original data facts by replacing `sym`i with $l5$ and $l7$ for both $i = 1$ and $i = 2$. However, MERGEDROID does not select any merge-and-replay point for $\texttt{p3.f} \,\|\, l5$ and $\texttt{p3.f} \,\|\, l7$ ($\texttt{q3.f} \,\|\, l5$ and $\texttt{q3.f} \,\|\, l7$) in `foo()` as it is difficult to determine which data facts share the same data abstraction at which program points, particularly in an IFDS-based algorithm. Our approach allows MERGEDROID to merge equivalent value-flows adequately, resulting in improved performance without sacrificing precision.

Our merge-and-replay approach incorporates context information in a symbolic activation statement, enabling MERGEDROID to prune unnecessary value flows and enhancing both efficiency and precision. In Figure 1b, during the analysis of `bar(s1, q1)` at line 25 where `q2` is the corresponding formal parameter of `q1`, MERGEDROID employs a call edge from $\texttt{q1.f} \,\|\, \texttt{sym1}$ to $\texttt{q2.f} \,\|\, \textbf{GAS}$. Here, **GAS** represents a globally unique symbolic activation statement introduced solely for computing summaries within MERGEDROID. This choice deviates from using a call edge from $\texttt{q1.f} \,\|\, \texttt{sym1}$ to $\texttt{q2.f} \,\|\, \texttt{sym1}$, since `sym1` originates at line 26 and does not match the calling context at line 25. This avoids a false leak reported by FLOWDROID at line 10 as follows. When `bar()` is analyzed, $\texttt{r2.f} \,\|\, \textbf{GAS}$ is found to be an alias of $\texttt{q1.f} \,\|\, \texttt{sym1}$ and thus tainted. While propagating $\texttt{r2.f} \,\|\, \textbf{GAS}$ into `foo()` at line 18 will indeed taint $\texttt{r3.f} \,\|\, \textbf{GAS}$, MERGEDROID will not activate it as **GAS** is not an activation statement in `foo()`. As a result, MERGEDROID will not report it as a false positive at line 10. Note that when `foo()` is analyzed due to line 18, $\texttt{q3.f} \,\|\, \textbf{GAS}$ is also tainted as $\texttt{q2.f} \,\|\, \textbf{GAS}$ is tainted. In this scenario, there is an extra overhead incurred while propagating $\texttt{q3.f} \,\|\, \textbf{GAS}$ within `foo()` (as it is not introduced by FLOWDROID). Interestingly, when analyzing `goo()` due to line 3, MERGEDROID replaces the call edge from $\texttt{p3.f} \,\|\, \ell$ to $\texttt{p4.f} \,\|\, \ell$ with a call edge from $\texttt{p3.f} \,\|\, \ell$ to $\texttt{p4.f} \,\|\, \textbf{GAS}$, where $\ell \in \{l5, l7\}$. This replacement effectively eliminates the redundant propagation of $\texttt{p4.f}$ within `goo()`.

GAS-related handling introduces occasional additional propagation and, in other instances, prevents redundant propagation. However, the substantial performance improvement resulting from this approach far surpasses the incurred overhead (as analyzed in Section IV-E and assessed in Section V).

Below we emphasize several noteworthy characteristics of MERGEDROID, which utilizes the *merge-and-replay* strategy:

- **Enhanced Efficiency and Precision.** MERGEDROID's merging of equivalent value flows within FLOWDROID, as illustrated in Figure 1, results in reduced alias queries and path edges, ultimately enhancing efficiency. By introducing context sensitivity into symbolic activation statements, MERGEDROID prunes spurious value flows in FLOWDROID, enhancing precision and eliminating false positives (e.g., as seen at line 10 of Figure 1), while also further boosting overall performance.
- **Multi-Threading Support.** MERGEDROID seamlessly operates within FLOWDROID's multi-threaded setup.
- **Simplicity.** MERGEDROID is conceptually simple and coded in about 400 lines of Java at its core.

III. PRELIMINARIES

We review the classic IFDS framework (Section III-A) and an IFDS-based taint analysis algorithm (Section III-B) to provide a foundation for understanding our *merge-and-replay* approach used by MERGEDROID, as presented in Section IV.

A. The IFDS Framework

The IFDS framework [6] addresses a special kind of data-flow problem, called *inter-procedural, finite, distributive, subset* (IFDS) problem, of which an instance *IP* is a quintuple $IP = (G^*, D, F, M, \sqcap)$, where $G^* = (N^*, E^*)$ is the *supergraph* of the program, D is a finite set of data-flow facts (i.e., data facts), $F \in 2^D \rightarrow 2^D$ is a set of distributive data-flow functions, $M : E^* \mapsto F$ is a map from the supergraph edges

Algorithm 1: The IFDS-based taint analysis algorithm employed by FLOWDROID [4]. The underlined statements are essential for facilitating context- and flow-sensitive taint analysis.

```
1  W_FW ← W_BW ← PathEdge_FW ← PathEdge_BW ← S_FW ← S_BW ← ∅
2  Prop(⟨main,0⟩ → ⟨s_main,0⟩, W_FW, PathEdge_FW)
3  ForwardAnalysis()
4  BackwardAnalysis()
5  function ForwardAnalysis()
6      while W_FW ≠ ∅ ∨ W_BW ≠ ∅ do
7          Wait until W_FW ≠ ∅ and pop ⟨m, d1⟩ → ⟨n, d2⟩ from W_FW
8          if n activates d2 then d2 = ActiveCopy(d2)
9          if n is a call node then
10             Let m′ be the method called at n and r be the return node of n
11             for d3 such that ⟨n, d2⟩ → ⟨s_m′, d3⟩ ∈ E#_FW do
12                 Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m′, d3⟩, E#_BW, PathEdge_BW, S_BW, W_BW)
13                 Prop(⟨m′, d3⟩ → ⟨s_m′, d3⟩, W_FW, PathEdge_FW)
14                 for ⟨m′, d3⟩ → ⟨e_m′, d4⟩ ∈ S_FW ∧ ⟨e_m′, d4⟩ → ⟨r, d5⟩ ∈ E#_FW do
15                     Prop(⟨m, d1⟩ → ⟨r, d5⟩, W_FW, PathEdge_FW)
16             for d3 such that ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E#_FW do
17                 Prop(⟨m, d1⟩ → ⟨r, d3⟩, W_FW, PathEdge_FW)
18         elif n = e_m then
19             Insert ⟨m, d1⟩ → ⟨n, d2⟩ into S_FW
20             for each callsite c that calls m do
21                 Let m″ (r) be the containing method (the return node) of c
22                 for ⟨m″, d3⟩ → ⟨c, d4⟩ ∈ PathEdge_FW ∧ ⟨c, d4⟩ → ⟨s_m, d1⟩ ∈ E#_FW
                       ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_FW do
23                     Prop(⟨m″, d3⟩ → ⟨r, d5⟩, W_FW, PathEdge_FW)
24         else
25             for ⟨n′, d3⟩ such that ⟨n, d2⟩ → ⟨n′, d3⟩ ∈ E#_FW do
26                 Prop(⟨m, d1⟩ → ⟨n′, d3⟩, W_FW, PathEdge_FW)
27                 if n is an assign node ∧ d3 ≠ d2 then
28                     d3′ = InactiveCopy(d3, n)
29                     Prop(⟨m, d1⟩ → ⟨n, d3′⟩, W_BW, PathEdge_BW)
30 function Prop(⟨m, d1⟩ → ⟨n, d2⟩, WorkList, PathEdge)
31     if ⟨m, d1⟩ → ⟨n, d2⟩ ∉ PathEdge then
32         Insert ⟨m, d1⟩ → ⟨n, d2⟩ into both PathEdge and WorkList
33 function BackwardAnalysis()
34     while W_BW ≠ ∅ ∨ W_FW ≠ ∅ do
35         Wait until W_BW ≠ ∅ and pop ⟨m, d1⟩ → ⟨n, d2⟩ from W_BW
36         if n is a call node then
37             Let m′ be the method called at n and r be the return node of n
38             for d3 such that ⟨n, d2⟩ → ⟨e_m′, d3⟩ ∈ E#_BW do
39                 Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m′, d3⟩, E#_FW, PathEdge_FW, S_FW, W_FW)
40                 Prop(⟨m′, d3⟩ → ⟨e_m′, d3⟩, W_BW, PathEdge_BW)
41                 for ⟨m′, d3⟩ → ⟨s_m′, d4⟩ ∈ S_BW ∧ ⟨s_m′, d4⟩ → ⟨r, d5⟩ ∈ E#_BW do
42                     Prop(⟨m, d1⟩ → ⟨r, d5⟩, W_BW, PathEdge_BW)
43             for d3 such that ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E#_BW do
44                 Prop(⟨m, d1⟩ → ⟨r, d3⟩, W_BW, PathEdge_BW)
45         elif n = s_m then
46             Insert ⟨m, d1⟩ → ⟨n, d2⟩ into S_BW
47             for each callsite c that calls m do
48                 Let m″ (r) be the containing method (the return node) of c
49                 for ⟨m″, d3⟩ → ⟨c, d4⟩ ∈ PathEdge_BW ∧ ⟨c, d4⟩ → ⟨e_m, d1⟩ ∈ E#_BW
                       ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_BW do
50                     Prop(⟨m″, d3⟩ → ⟨r, d5⟩, W_BW, PathEdge_BW)
51         else
52             for ⟨n′, d3⟩ such that ⟨n, d2⟩ → ⟨n′, d3⟩ ∈ E#_BW do
53                 Prop(⟨m, d1⟩ → ⟨n′, d3⟩, W_BW, PathEdge_BW)
54                 if n is an assign node ∧ d3 ≠ d2 then
55                     Prop(⟨m, d1⟩ → ⟨n, d3⟩⟩, W_FW, PathEdge_FW)
56 function Inject(⟨m, d1⟩ → ⟨c, d2⟩, ⟨m′, d3⟩, E#, PathEdge, S, W)
57     Insert ⟨m, d1⟩ → ⟨c, d2⟩⟩ into PathEdge
58     Let r be the return node of c
59     for ⟨m′, d3⟩ → ⟨n′, d4⟩ ∈ S ∧ ⟨n′, d4⟩ → ⟨r, d5⟩ ∈ E# do
60         Prop(⟨m, d1⟩ → ⟨r, d5⟩, W, PathEdge)
```

to data-flow functions, and the meet operator $\sqcap$ is either union or intersection (depending on the problem modeled).

The supergraph G^* consists of a set of control flow graphs (CFGs), one for each method. For a method m, its CFG G_m has a unique *start* node $s_m \in N^*$ and *exit* node $e_m \in N^*$. A callsite is represented by a *call* node $c \in N^*$ and a *return* node by $r \in N^*$. The remaining nodes, termed *normal* nodes, encompass statements and predicates in the usual manner. Examples include *assign* nodes for representing assignment statements. Edges in E^* are classified into four kinds: *call edges* (connecting a call node to a start node), *return edges* (connecting an exit node to a return node), *call-to-return edges* (connecting a call node to a return node), and *normal edges* (connecting normal nodes).

Reps et al. [6] reduce the IFDS problem to a graph reachability problem on an *exploded supergraph* $G^{\#}_{IP} = (N^{\#}, E^{\#})$ transformed from the supergraph G^*, where $N^{\#} = N^* \times (D \cup \{\mathbf{0}\})$ and $E^{\#} = \{\langle n_1, d_1\rangle \to \langle n_2, d_2\rangle \mid n_1 \to n_2 \in E^*, f = M(n_1, n_2), d_2 \in f(d_1)\}$. Note that $f \in F$ is the data-flow function of the edge $n_1 \to n_2 \in E^*$, and $\mathbf{0}$ is a special fact that allows new facts to be generated at some program points. The graph reachability problem is tackled through an efficient tabulation algorithm, functioning as a worklist algorithm. It begins from an initial path edge $\langle s_{main}, \mathbf{0}\rangle \to \langle s_{main}, \mathbf{0}\rangle$, with $main$ signifying the program's main entry. This algorithm accumulates additional path edges until a fixed point is achieved. Each path edge $\langle s_m, d_1\rangle \to \langle n, d_2\rangle$ symbolizes a feasible path suffix in $G^{\#}IP$ from $\langle s_{main}, \mathbf{0}\rangle$ to $\langle n, d_2\rangle$. For comprehensive details, we refer to [6, Figure 3].

B. IFDS-Based Taint Analysis Algorithm

The IFDS-based taint analysis algorithm utilized by FLOWDROID [4] (Algorithm 1) has a forward analysis (lines 5-29) and a backward analysis (lines 33-55), both of which are IFDS-based and executed iteratively. To better understand the algorithm, we introduce some notations below.

In the forward (backward) IFDS analysis (solver), we use W_{FW}, $PathEdge_{\text{FW}}$, S_{FW}, and $E^{\#}_{\text{FW}}$ (W_{BW}, $PathEdge_{\text{BW}}$, S_{BW}, and $E^{\#}_{\text{BW}}$) to represent the worklist, path edges, end summaries (a variation of traditional summaries [6] extended in [13]), and

exploded supergraph edges. A path edge can be processed by both analyses, which swap entry and exit nodes, so we use a variant form of path edge $\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle$, where m is a method, n is a statement, and d_1 and d_2 are data facts. A data fact d is represented as $abs \,\|\, as$, where abs denotes its data abstraction and as is its activation statement. Furthermore, d is indicative of being a taint (i.e., *active*) when $as = null$, while it remains *inactive* otherwise. To facilitate these operations, two auxiliary functions are used. $\texttt{ActiveCopy}(d)$ generates an active copy of d by setting $as = null$, and $\texttt{InactiveCopy}(d, n)$ returns d if it is inactive, otherwise, it produces an inactive copy of d with its activation statement replaced by n.

In Algorithm 1, the forward (lines 5-29) and backward (lines 33-55) IFDS-based analyses detect taints and aliases, respectively. They are initialized in lines 1-4 and only terminate when both $W_{\texttt{FW}}$ and $W_{\texttt{BW}}$ are empty (lines 6 and 34).

During the forward analysis, a new path edge $\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle$ is retrieved from $W_{\texttt{FW}}$ (line 7), and if possible, d_2 is activated (line 8). In FLOWDROID [4], a statement n can activate a data fact d if n is the activation statement of d or a callsite that transitively invokes the method where the activation statement of d is located. Lines 9-26 follow the conventional approach of the classic IFDS algorithm [6], with the exception of line 12. In particular, lines 9-17 (18-23) manage the inter-procedural data flows entering (exiting) a method, while lines 24-26 manage the intra-procedural data flows within a method. In the case where statement n represents an assignment (lines 27-29), a path edge $\langle m, d_1\rangle \rightarrow \langle n, d_3'\rangle$ is inserted into the backward analysis to identify aliases of d_3, where d_3' denotes an inactive data fact produced by $\texttt{InactiveCopy}$ (d_3, n).

Moving to the backward analysis, lines 36-53 are also standard except for line 39. Note that in the backward analysis, e_m is considered as the start node, while s_m is the exit node, in contrast to the forward analysis. When analyzing an assignment statement, a new path edge $\langle m, d_1\rangle \rightarrow \langle n, d_3\rangle$ is injected into the forward analysis to find data facts tainted by d_3 (lines 54-55). Despite the current inactivity of d_3, these tainted data facts may become activated in the future (line 8).

In FLOWDROID [4], a call statement can generate new data facts that need to be propagated to the other IFDS solver. To handle this, a path edge $\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle$ is injected at the call statement into the other solver (lines 12 and 39).

IV. MERGEDROID: OUR APPROACH

We outline our *merge-and-replay* algorithm in Algorithm 2, extending Algorithm 1. Here, each line l is replaced by line la, and optionally lb for $11 \leq l \leq 50$, with significant changes highlighted in light gray. The "..." between lines 15a and 22a (42a and 49a) indicates the omission of lines 16–21 (lines 43–48) from Algorithm 1. The notations used in Algorithm 2 are introduced in Section IV-A. We explain where and how equivalent value-flows are merged in Section IV-B, and how to replay (unmerge) merged value-flows in Section IV-C and prune spurious ones in Section IV-D. Finally, we provide an overhead analysis and discuss design choices in Section IV-E.

A. Notations

Given a data fact $d = abs \,\|\, as$, $\texttt{DataAbstraction}(d)$ and $\texttt{ActivationStmt}(d)$ return its data abstraction abs and its activation statement as, respectively. Given a symbolic activation statement $sym = \langle abs, caller, callee\rangle$, $\texttt{Context}(sym) = \langle caller, callee\rangle$ returns its context. In addition, two functions are used for recovering value flows for sym. *Symb2Reps* maps sym to the set of activation statements represented by sym. *SymbolIncoming* maps sym to the set of path-edge and abstraction pairs $\langle\langle caller, d_1\rangle \rightarrow \langle c, d_2\rangle, abs\rangle$, where $\langle caller, d_1\rangle \rightarrow \langle c, d_2\rangle$ is a path edge at the callsite c, sym is the symbolic activation statement of d_2, $\langle caller, callee\rangle = \texttt{Context}(sym)$, and abs is a data abstraction of a data fact d_3 such that $\langle c, d_2\rangle \rightarrow \langle s_{callee}, d_3\rangle \in E^{\#}_{\texttt{FW}}$ is an forward exploded supergraph edge processed.

GAS is a globally unique activation statement introduced for pruning value flows. We temporarily exclude its related lines (underlined) from Algorithm 2 in Section IV-B and Section IV-C, as they will be covered in Section IV-D.

B. Merging Equivalent Value-flows Symbolically

In the backward pass, equivalent data facts with the same data abstraction abs at any program point can be merged to avoid separate propagation. `Symbolize()` (lines 70-77) creates a special symbolic activation statement sym to represent their concrete activation statements and returns a symbolic data fact with abs decorated by sym (line 77) for propagation. The concrete activation statements of sym are stored in *Symb2Reps*(sym) for future use (lines 74-75).

Determining which set of data facts should have their activation statements symbolized and at which program points is a challenge because multiple data facts sharing the same data abstraction can occur at unpredictable program points, especially for the IFDS algorithm [6]. Symbolizing every data abstraction at every program point would be impractical due to memory overhead. Instead, we only perform symbolization at callsites for the set of data facts propagated from the same callee method. Furthermore, we merge data facts arriving at different callsites in the same caller method from the same callee method. As shown at lines 41b and 49b of Algorithm 2, we utilize `OnReturnFlow()`, which calls `Symbolize()` (line 103), to acquire a symbolic data fact for ongoing propagation. In alignment with our design decision, a symbolic activation statement encompasses a data abstraction abs, a caller method $caller$, and a callee method $callee$ (line 73). We will discuss some other design choices in Section IV-E shortly. Line 76 is concerned with replaying value flows and will be explained further in Section IV-C.

C. Replaying Merged Value-flows Concretely

In the forward pass, activating data facts containing symbolic activation statements poses a challenge in MERGEDROID (Algorithm 1). Symbolic activation statements result from merging equivalent value flows in the backward pass and are propagated through path edges in the forward analysis. In Algorithm 1, inactive data facts are activated at line 8 upon

Algorithm 2: The *merge-and-replay* algorithm in MERGEDROID (built on top of Algorithm 1). The lines shaded in light gray (underlined) indicate part of the algorithm responsible for value-flow consolidation (value-flow pruning).

```
   function ForwardAnalysis()
11a  for d3 such that ⟨n, d2⟩ → ⟨s_m', d3⟩ ∈ E#_FW do
11b    for d'3 ∈ Concretize(⟨m, d1⟩ → ⟨n, d2⟩, m', d3) do
12a      Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m', d'3⟩, E#_BW, PathEdge_BW, S_BW, W_BW)
13a      Prop(⟨m', d'3⟩ → ⟨s'_m, d'3⟩, W_FW, PathEdge_FW)
14a      for ⟨m', d'3⟩ → ⟨e_m', d4⟩ ∈ S_FW ∧ ⟨e_m', d4⟩ → ⟨r, d5⟩ ∈ E#_FW do
14b        d'5 = AttachActivationStmt(d5, d2)
15a        Prop(⟨m, d1⟩ → ⟨r, d'5⟩, W_FW, PathEdge_FW)
     ...
22a  for ⟨m'', d3⟩ → ⟨c, d4⟩ ∈ PathEdge_FW ∧ ⟨c, d4⟩ → ⟨s_m, d1⟩ ∈ E#_FW
       ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_FW do
22b    d'5 = AttachActivationStmt(d5, d4)
23a    Prop(⟨m'', d3⟩ → ⟨r, d'5⟩, W_FW, PathEdge_FW)
   function BackwardAnalysis()
38a  for d3 such that ⟨n, d2⟩ → ⟨e_m', d3⟩ ∈ E#_BW do
38b    d'3 = DataAbstraction(d3) || GAS
39a    Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m', d'3⟩, E#_FW, PathEdge_FW, S_FW, W_FW)
40a    Prop(⟨m', d'3⟩ → ⟨e_m', d'3⟩, W_BW, PathEdge_BW)
41a    for ⟨m', d'3⟩ → ⟨s_m', d4⟩ ∈ S_BW ∧ ⟨s_m', d4⟩ → ⟨r, d5⟩ ∈ E#_BW do
41b      d'5 = OnReturnFlow(d5, d2, m, m')
42a      Prop(⟨m, d1⟩ → ⟨r, d'5⟩, W_BW, PathEdge_BW)
     ...
49a  for ⟨m'', d3⟩ → ⟨c, d4⟩ ∈ PathEdge_BW ∧ ⟨c, d4⟩ → ⟨e_m, d1⟩ ∈ E#_BW
       ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_BW do
49b    d'5 = OnReturnFlow(d5, d4, m'', m)
50a    Prop(⟨m'', d3⟩ → ⟨r, d'5⟩, W_BW, PathEdge_BW)
61 function OnActivationStmtAdded(sym, as)
62   ⟨m, m'⟩ = Context(sym)    // m is caller, m' is callee
63   for ⟨⟨m, d1⟩ → ⟨c, d2⟩, abs⟩ ∈ SymbolIncoming(sym) do
64     d3 = abs || as
65     Prop(⟨m', d3⟩ → ⟨s_m', d3⟩, W_FW, PathEdge_FW)
66     Let r be the return node of c
67     for ⟨m', d3⟩ → ⟨e_m', d4⟩ ∈ S_FW ∧ ⟨e_m', d4⟩ → ⟨r, d5⟩ ∈ E#_FW do
68       Prop(⟨m, d1⟩ → ⟨r, d5⟩, W_FW, PathEdge_FW)

69 SymbolIncoming = Symb2Reps = {}
70 function Symbolize(d_ret, caller, callee)
71   as = ActivationStmt(d_ret)
72   abs = DataAbstraction(d_ret)
73   sym = ⟨abs, caller, callee⟩    // Symbolic activation stmt
74   if as ∉ Symb2Reps(sym) then
75     Symb2Reps(sym) ∋ as
76     OnActivationStmtAdded(sym, as)
77   return abs || sym
78 function Concretize(⟨caller, d1⟩ → ⟨c, d2⟩, callee, d3)
79   if d3 is active then return {d3}
80   as = ActivationStmt(d3)
81   if as is a concrete statement or as = GAS then return {d3}
82   assert as is a symbolic activation statement
83   sym = as                                  // Rename variable
84   reps = {}                          // Set of represented facts
85   abs = DataAbstraction(d3)
86   if Context(sym) = ⟨caller, callee⟩ then
87     SymbolIncoming(sym) ∋ ⟨⟨caller, d1⟩ → ⟨c, d2⟩, abs⟩
88     for v ∈ Symb2Reps(sym) do
89       reps ∋ abs || v
90   else
91     reps ∋ abs || GAS
92   return reps
93 function AttachActivationStmt(d_ret, d_call)
94   u = ActivationStmt(d_ret)
95   v = ActivationStmt(d_call)
96   if u = GAS then
97     return DataAbstraction(d_ret) || v
98   return d_ret
99 function OnReturnFlow(d_ret, d_call, caller, callee)
100  u = ActivationStmt(d_ret)
101  if u = GAS then
102    return AttachActivationStmt(d_ret, d_call)
103  return Symbolize(d_ret, caller, callee)
```

passing through their activation statements or at any callsite that transitively invokes methods containing these activation statements. However, activating data facts with symbolic activation statements requires careful consideration due to their potential representation of a set of activation statements, which may include other symbolic activation statements.

To address this challenge, we intuitively replay value flows during the forward analysis, just before data facts containing symbolic activation statements are propagated into a method (as indicated in line 11b and `Concretize()` (lines 78-92)). For active data facts or those with non-symbolic activation statements, we propagate them normally (lines 79-81). For a data fact, say, d_3, with a symbolic activation statement sym (line 83) and data abstraction abs (line 85), we use the set of concrete activation statements recorded in *Symb2Reps*(sym) (represented by sym) to decorate abs and create new data facts for propagation into the method being analyzed (lines 88-89). For any activation statements not yet recorded in *Symb2Reps*(sym) (due to IFDS being distributive [6]), the incoming path edge $\langle caller, d_1\rangle \rightarrow \langle c, d_2\rangle$ and abs are saved into *SymbolIncoming*(sym) (line 87). When a new activation statement u is later recorded in *Symb2Reps*(sym) (line 75), `OnActivationStmtAdded` is called (line 76) to complete the value-flow replaying (lines 61-68).

Theorem 1. (Preservation of Precision). Algorithm $\mathcal{A}$, obtained by deleting the underlined statements in Algorithm 2, replacing d'_3 with d_3 in lines 39a-41a, and d'_5 with d_5 in lines 14b-23a, yields the same data leak results as Algorithm 1.

Proof Sketch. Follows from the fact that every merged data flow fact is replayed before it reaches its activation points. □

D. Pruning Spurious Value flows

In addition to enhancing the performance of IFDS-based taint analysis, MERGEDROID further improves its precision by removing false-positive value flows. This is achieved by leveraging a key observation described below.

Observation 1. Algorithm 1's backward pass is only triggered on-demand in certain contexts, highlighting the need for context-sensitive activation statements that allow for specific data facts to be propagated under certain contexts.

Activation statements in FLOWDROID (Algorithm 1) are context-insensitive, causing spurious value flows that degrade not only its efficiency but also its precision. Observation 1 highlights the importance of context-sensitive activation statements to propagate data facts with specific activation statements only in certain contexts. Such context information has not been exploited in previous taint analysis research.

Efficiently maintaining activation statement context information without excessive overhead is challenging. A cloning-based approach similar to pointer analysis [14]–[16], utilizing k context elements (like callsites in callsite sensitivity [17]–[20] and allocation sites in object sensitivity [21], [22]) to differentiate under k-limited calling contexts (typically small, e.g., 2), would significantly increase time and memory overheads. This is due to the necessity of managing a large number of data facts in an IFDS-based taint analysis framework. Instead, we have opted to encode context information implicitly in symbolic activation statements. This approach seamlessly aligns with the IFDS algorithm [6] and achieves precision by analyzing activation statements with full context sensitivity.

In Algorithm 2, the underlined statements leverage context information within symbolic activation statements to prevent spurious value flows during the handling of call statements in both the forward and backward passes.

During the forward analysis, value-flow replaying is performed (lines 87-89) only when the context of the symbolic activation statement sym matches the current calling context $\langle caller, callee\rangle$ (line 86). Otherwise, a new data fact is formed using **GAS** (line 91). Since sym is always uniquely identified by $\langle caller, callee\rangle$ (line 73), all activation statements are therefore analyzed with full callsite-based context sensitivity within the IFDS framework, as intended.

During the backward analysis, a newly established data fact through a return edge at a callsite (line 38a) is always formed by replacing the activation statement with **GAS** (line 38b). This propagation of a **GAS**-decorated data fact aims to compute reusable summaries in callee methods. If a **GAS**-decorated data fact is returned from an exit statement (entry statement) to a callsite at lines 14b and 22b (lines 41b and 49b) during the forward (backward) analysis, **GAS** is replaced with the activation statement of the corresponding data fact at the callsite (line 97) using `AttachActivationStmt()` (lines 93-98), either directly (lines 14b and 22b) or indirectly (lines 41b, 49b, and 101-102). Overall, **GAS**-decorated data facts aid in pruning value flows, enhancing precision by reducing false positives (e.g., the false leak reported by FLOWDROID at line 10 of Figure 1), and improving efficiency by avoiding spurious propagation. However, they may introduce additional propagation (e.g., `q3.f` ‖ **GAS** in `foo()` in Figure 1). The overall benefits significantly outweigh the incurred overhead.

Theorem 2. (Improvement of Precision). Algorithm 2 is designed to generate a subset of the leak warnings reported by Algorithm 1, which may not be strictly smaller in all cases.

Proof Sketch. Follows from Algorithm 2 pruning only spurious value-flows from mismatched contexts (line 86). □

E. Overhead Analysis and Design Choices

We explore the time and space overheads introduced by Algorithm 2 in comparison to Algorithm 1.

To maintain *SymbolIncoming* and *Symb2Reps*, additional space is introduced. The worst-case space required for *Symb2Reps* is $O(c \cdot |P| \cdot |A|)$, where c is the maximum number of merged activation statements per symbolic activation statement, $|P|$ is the caller-callee pairs, and $|A|$ is the maximum abstractions per method. For *SymbolIncoming*, it is $O(|E^{call}| \cdot |D|^2 \cdot |A|)$, where $|E^{call}|$ is total call edges and $|D|$ is total data facts. Given that $|P| < |E^{call}|$ and c is small in practice, the overall space overhead is $O(|E^{call}| \cdot |D|^2 \cdot |A|)$.

The time complexities of Algorithms 1 and 2 match that of the IFDS algorithm [6], i.e., $O(|E^*| \cdot |D|^3)$. Therefore, we only estimate the time overhead introduced by *merge-and-replay* (Algorithm 2). This overhead originates from three sources. First, the operations for maintaining *SymbolIncoming* and *Symb2Reps* (lines 63-68, 75, and 87-89) collectively require $O(|E^{call}| \cdot |D|^2 \cdot |A|)$, significantly smaller than $O(|E^*| \cdot |D|^3)$. Second, to prevent data races in a multi-threaded environment, both *SymbolIncoming* and *Symb2Reps* utilize a lock-based data structure, which may result in minor additional time overhead. Typically, this overhead is negligible, as it is a small constant related to the number of operations on these structures. Finally, while the path edges created by **GAS**-decorated data facts intend to compute reusable summaries, not all are reused in the analysis. The redundant propagation they introduce could consume $O(|E^*| \cdot |A|^3 + |E^{call}| \cdot |A|(|D|^2 - |A|^2))$ in extreme cases, which is rare in practice and overly conservative.

Finally, we delve into the design choices concerning the representation of symbolic activation statements. As described in Section IV-B, when a set of data facts sharing the same data abstraction within a callee are propagated to their corresponding callsites within a given caller, they are merged into a single data fact utilizing a symbolic activation statement. According to Theorem 1, encoding either a callsite or a caller as part of the context information for a symbolic activation statement does not compromise the precision-preserving property of our approach. While encoding a callsite permits the merging of activation statements from various callee methods of a polymorphic call, polymorphic calls constitute only a small proportion of the overall calls in most programs. Therefore, in our current implementation, we have opted to encode a caller instead of a callsite, to avoid introducing unnecessary memory overhead without substantial performance gains in practice.

V. EVALUATION

We demonstrate the significant performance benefits and precision improvements of our *merge-and-replay* algorithm by comparing MERGEDROID with FLOWDROID. Our evaluation aims to answer the following three research questions:

TABLE I: Comparing the performance of FLOWDROID (FD) and MERGEDROID (MD). The apps are categorized based on their sources and are ordered in increasing order of FLOWDROID's analysis time. OoT stands for Out of Time.

Group	App	Version	Analysis Time (s)		Memory Usage (GB)		#PathEdges (M)		#Leaks	
			FD	MD	FD	MD	FD	MD	FD	MD
From [9]	com.ilm.sandwich	2.2.4f	2	1 (2.0×)	0.8	0.4 (2.0×)	0.5	0.3 (1.7×)	9	9 (0.0%)
	com.github.yeriomin.dumbphoneassistant	0.5	3	1 (3.0×)	0.9	0.5 (1.8×)	1.0	0.2 (5.0×)	2	2 (0.0%)
	com.poupa.vinylmusicplayer	0.20.1	4	1 (4.0×)	1.2	1.1 (1.1×)	0.4	0.1 (4.0×)	5	4 (20.0%)
	dk.jens.backup	0.3.4	4	1 (4.0×)	0.9	0.6 (1.5×)	0.5	0.2 (2.5×)	3	3 (0.0%)
	org.csploit.android	1.6.5	7	4 (1.8×)	1.9	1.5 (1.3×)	0.3	0.2 (1.5×)	1	1 (0.0%)
	com.kunzisoft.keepass.libre	2.5.0.0beta18	8	2 (4.0×)	0.9	0.8 (1.1×)	2.7	0.6 (4.5×)	6	3 (50.0%)
	org.materialos.icons	2.1	21	11 (1.9×)	2.1	1.3 (1.6×)	6.9	3.4 (2.0×)	3	3 (0.0%)
	com.app.Zensuren	1.21	22	13 (1.7×)	2.4	1.7 (1.4×)	8.0	4.7 (1.7×)	9	9 (0.0%)
	org.decsync.sparss.floss	1.13.4	27	35 (0.8×)	3.5	3.5 (1.0×)	5.9	5.3 (1.1×)	25	24 (4.0%)
	de.schildbach.oeffi	10.5.3-google	34	8 (4.2×)	2.1	0.9 (2.3×)	6.6	1.8 (3.7×)	8	7 (12.5%)
	org.secuso.privacyfriendlytodolist	2.1	48	5 (9.6×)	6.3	1.0 (6.3×)	17.7	1.5 (11.8×)	4	2 (50.0%)
	name.myigel.fahrplan.eh17	1.33.16	81	3 (27.0×)	3.0	0.3 (10.0×)	8.0	0.2 (40.0×)	4	2 (50.0%)
	com.emn8.mobilem8.nativeapp.bk	5.0.10	182	51 (3.6×)	2.6	0.8 (3.2×)	3.5	0.9 (3.9×)	25	25 (0.0%)
	com.microsoft.office.word	16.0.11425.20132	204	121 (1.7×)	5.4	2.0 (2.7×)	12.1	3.4 (3.6×)	13	8 (38.5%)
	com.vonglasow.michael.satstat	3.3	310	28 (11.1×)	25.8	2.9 (8.9×)	127.2	10.2 (12.5×)	7	6 (14.3%)
	com.adobe.reader	19.2.1.9183	742	74 (10.0×)	6.1	2.1 (2.9×)	11.5	1.4 (8.2×)	20	19 (5.0%)
	org.totschnig.myexpenses	3.0.1.2	746	31 (24.1×)	53.1	2.9 (18.3×)	247.3	6.5 (38.0×)	16	15 (6.2%)
	com.igisw.openmoneybox	3.2.2.10	1222	276 (4.4×)	83.7	21.4 (3.9×)	407.2	106.8 (3.8×)	9	8 (11.1%)
	com.ichi2.anki	2.8.4	OoT	877	-	13.9	-	51.2	-	21
	org.openpetfoodfacts.scanner	2.9.8	OoT	18	-	2.2	-	6.6	-	4
From [10]	org.gateshipone.odyssey	1.1.18	8	3 (2.7×)	1.2	0.4 (3.0×)	2.9	0.9 (3.2×)	7	3 (57.1%)
	com.alfray.timeriffic	1.09.05	21	4 (5.2×)	3.2	1.6 (2.0×)	7.4	1.1 (6.7×)	15	11 (26.7%)
	com.github.axet.callrecorder	1.17.13	65	9 (7.2×)	7.2	1.0 (7.2×)	21.2	2.8 (7.6×)	10	6 (40.0%)
	org.secuso.privacyfriendlyweather	2.1.1	371	5 (74.2×)	28.5	0.4 (71.2×)	112.5	1.2 (93.8×)	5	5 (0.0%)
	com.genonbeta.TrebleShot	1.4.2	385	56 (6.9×)	7.9	1.9 (4.2×)	30.2	3.9 (7.7×)	3	2 (33.3%)
	org.fdroid.fdroid	1.8-alpha0	477	103 (4.6×)	4.2	2.4 (1.8×)	7.3	2.8 (2.6×)	26	14 (46.2%)
	com.github.axet.bookreader	1.12.14	856	249 (3.4×)	66.5	21.0 (3.2×)	372.6	115.5 (3.2×)	1	1 (0.0%)
	com.kanedias.vanilla.metadata	1.0.4	1035	11 (94.1×)	57.0	1.7 (33.5×)	193.2	3.3 (58.5×)	1	1 (0.0%)
	bus.chio.wishmaster	1.0.2	2894	36 (80.4×)	87.7	1.7 (51.6×)	391.0	3.7 (105.7×)	9	7 (22.2%)
	org.lumicall.android	1.13.1	2904	100 (29.0×)	48.9	1.7 (28.8×)	247.9	5.2 (47.7×)	7	5 (28.6%)
	nya.miku.wishmaster	1.5.0	4551	33 (137.9×)	117.1	1.4 (83.6×)	562.3	3.7 (152.0×)	6	5 (16.7%)
	de.k3b.android.androFotoFinder	0.8.0.191021	OoT	268	-	5.4	-	18.2	-	12
	fr.gouv.etalab.mastodon	3.21.2	OoT	423	-	8.3	-	37.9	-	18
F-Droid	com.icecondor.nest	20150402	22	1 (22.0×)	2.8	0.4 (7.0×)	5.5	0.1 (55.0×)	5	4 (20.0%)
	com.dimowner.audiorecorder	0.9.26	53	3 (17.7×)	1.7	1.2 (1.4×)	3.6	0.2 (18.0×)	5	4 (20.0%)
	me.austinhuang.caweather	2.4	238	6 (39.7×)	20.6	0.8 (25.8×)	93.0	2.0 (46.5×)	8	6 (25.0%)
	com.activitymanager	4.1.4	460	6 (76.7×)	27.6	0.8 (34.5×)	114.3	1.6 (71.4×)	3	2 (33.3%)
	de.deftk.openww.android	0.4.3	640	9 (71.1×)	26.5	1.3 (20.4×)	80.0	2.6 (30.8×)	7	6 (14.3%)
	net.sourceforge.opencamera	1.49.2	OoT	216	-	19.9	-	74.8	-	3
	com.fastaccess.github.libre	4.6.7	OoT	10	-	1.6	-	1.2	-	1
Mean	-	-	-	9.0×	-	5.2×	-	9.7×	-	19.2%

- **RQ1.** Is MERGEDROID more precise?
- **RQ2.** Is MERGEDROID faster?
- **RQ3.** Is MERGEDROID more memory-efficient?

Implementation. We have built MERGEDROID on top of a recent FLOWDROID revision (`d8c80ac`). With about 400 lines at its core, our implementation is compact and efficient, benefiting from our streamlined algorithms and key insights. MERGEDROID is open-source and can be accessed at https://www.cse.unsw.edu.au/~corg/MergeDroid/.

Benchmark Selection. Due to the absence of standardized benchmarks, we have considered a set of 58 apps from previous studies, comprising 40 from [9] and 18 from [10], which are carefully selected by their authors for comparing their tools with FLOWDROID. However, we have excluded 25 apps from this set, including 3 apps that caused FLOWDROID to crash, 3 apps with no source or sink definitions due to using a newer version of FLOWDROID, 3 apps used in both [9] and [10] but with different versions (we used their newer versions), 14 apps that could be analyzed by FLOWDROID in 3 seconds, and 2 apps that were unscalable within a 3-hour budget by both FLOWDROID and MERGEDROID due to running out of time or memory. We have updated `fr.gouv.etalab.mastodon`

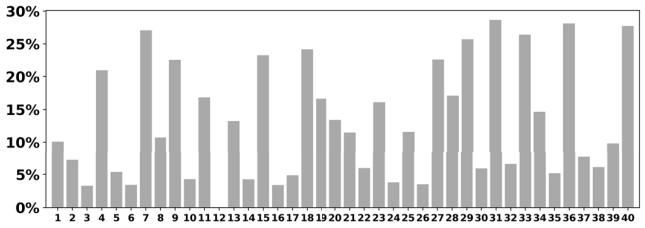


Fig. 2: Percentage of symbolic activation statements that merge more than one activation statement (symbolic or concrete) in each app, identified by its ordinal number in Table I.

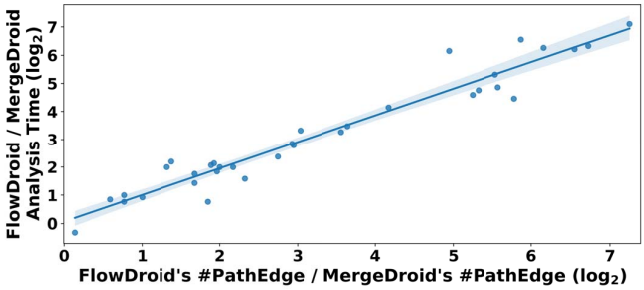


Fig. 3: Comparing MERGEDROID and FLOWDROID by correlating the reduction in #PathEdge with that in analysis time.

to its latest version as the one used in [10] is no longer available. In addition to the remaining 33 apps (20 from [9] and 13 from [10]), we have selected 7 apps from F-Droid [12] to increase the total number of apps to 40.

Experimental Setting. Experiments were conducted on an Ubuntu 20.04.5 LTS (Focal Fossa) machine with 8 CPU cores (16 processors) and 512GB RAM. The maximum heap size for the JVM was set to 256GB (`-Xmx`). Both FLOWDROID and MERGEDROID utilized 16 threads for their IFDS-based taint analysis, with a 3-hour time budget using the `-dt` option. Analysis of multiple dex files in Android apps was enabled via `--mergedexfiles`. The maximum callback chain depth was specified with `-md`, and path reconstruction mode was set to `precise` for detailed leak reports with `-pr`. Default values in FLOWDROID were applied to other options, including the default access path length of 5 and source/sink definitions.

Results. Table I presents our main results across four dimensions: analysis time (Columns 4-5), maximum memory consumption during analysis (Columns 6-7), number of path edges processed by the forward and backward solvers (Columns 8-9), and number of reported leaks (Columns 10-11). For each app, the analysis time, memory usage, and path edges (#PathEdges) represent averages drawn from three runs. The final row displays average values, with the first three being geometric means and the last being the arithmetic mean.

A. RQ1: Precision

We have validated the correctness of MERGEDROID using two benchmark suites with established ground truth: DROIDBENCH (the FLOWDROID companion benchmark suite) [23] and TAINTBENCH (a real-world malware benchmark suite) [24]. Additionally, the theoretical assurances provided by Theorems 1 and 2 reinforce its reliability. MERGEDROID has demonstrated success in passing all test cases that FLOWDROID has successfully handled. Our primary focus now lies in assessing the enhanced precision of MERGEDROID over FLOWDROID in terms of identifying false positive leaks.

As shown in Columns 10-11 of Table I, MERGEDROID achieves a significant reduction in the number of reported leaks on 24 out of 34 apps (that are analyzed scalably by both tools), indicating a significant improvement in precision compared to FLOWDROID. On average, MERGEDROID reduces the number of leak warnings reported by FLOWDROID by 19.2% (in percentage) for the 34 apps that can be analyzed by both tools, with a maximum reduction of 57.1% observed on `org.gateshipone.odyssey`. This precision improvement is attributed to the use of context information of activation statements, as explained in Section IV.

B. RQ2: Speedups

In Table I, Columns 4-5 highlight MERGEDROID's improvements in both scalability and efficiency compared to FLOWDROID. MERGEDROID successfully scales to analyze all 40 apps. Additionally, for the 6 apps where FLOWDROID exceeds its time limit, MERGEDROID completes the analysis within 1812 seconds. The speedups of MERGEDROID over FLOWDROID for the remaining 34 apps range from 0.8× to 137.9× with an average of 9.0×. Typically, MERGEDROID is more effective for large apps as they present more opportunities for applying our merge-and-replay strategy. The three largest speedups are observed on `nya.miku.wishmaster` (137.9×), `com.kanedias.vanilla.metadata` (94.1×), and `bus.chio.wishmaster` (80.4×). For smaller apps, MERGEDROID is generally faster than FLOWDROID, but the speedups are relatively smaller.

Below we analyze the reasons behind these speedups.

In Figure 2, about 13.0% of symbolic activation statements on average can merge more than one activation statement (either symbolic or concrete), demonstrating MERGEDROID's effectiveness in consolidating equivalent value flows. Additionally, upon invoking `Concretize()` in Algorithm 2, the condition $\texttt{Context}(sym) = \langle caller, callee \rangle$ (line 86) fails to hold for an average of 54.5%, underscoring the successful pruning of numerous spurious value flows (line 90). In the case of `name.myigel.fahrplan.eh17`, MERGEDROID achieves an impressive speedup of 27.0× over FLOWDROID (Table I). While effective mergings for activation statements are relatively infrequent (Figure 2), spurious value-flow prunings occur 41.7% of the time (line 90). Consequently, MERGEDROID significantly reduces the number of path edges processed by FLOWDROID by 40.0×, leading also to a reduction in the operations required for reallocating and rehashing the *PathEdge* data structure during the analysis [8], [25]. Both reductions contribute to MERGEDROID's achieved speedup.

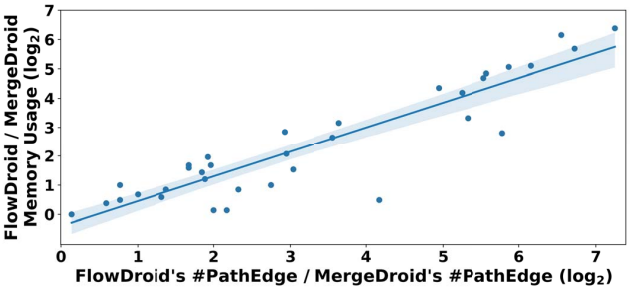


Fig. 4: Comparing MERGEDROID and FLOWDROID by correlating the reduction in #PathEdge with that in memory usage.

We have further compared MERGEDROID with FLOWDROID by examining the relationship between the reduction in path edges and analysis time. Figure 3 illustrates this correlation, where the ratios $\frac{\text{FLOWDROID}'s\ \#PathEdge}{\text{MERGEDROID}'s\ \#PathEdge}$ and $\frac{\text{FLOWDROID}'s\ Analysis\ Time}{\text{MERGEDROID}'s\ Analysis\ Time}$ show a strong positive correlation (Pearson correlation coefficient: 0.93). This underscores that the speedups achieved by MERGEDROID are attributed to the effectiveness of our *merge-and-replay* algorithm.

In our evaluation, we noted a single instance of slowdown on `org.decsync.sparss.floss`. Despite processing 10.2% fewer path edges than FLOWDROID, MERGEDROID experienced a 29.6% (8-second) slowdown in analyzing this app. This minor slowdown could be attributed to the overhead of MERGEDROID, as discussed in Section IV-E.

C. RQ3: Memory Requirements

MERGEDROID also significantly reduces the memory requirements of FLOWDROID, as indicated by the memory usage reduction factor for each app given in parentheses in Columns 6-7. We calculate the maximum amount of memory consumed by each app using the same Java Runtime APIs as in FLOWDROID. For all the 34 apps analyzed by FLOWDROID, MERGEDROID uses less memory. The memory usage ratios of FLOWDROID over MERGEDROID range from 1.0× (`org.decsync.sparss.floss`) to 83.6× (`nya.miku.wishmaster`) with an average of 5.2×.

The memory reduction achieved by MERGEDROID in comparison to FLOWDROID is attributed to the decrease in the number of processed path edges, as depicted in Figure 4. The correlation between $\frac{\text{FLOWDROID}'s\ \#PathEdge}{\text{MERGEDROID}'s\ \#PathEdge}$ and $\frac{\text{FLOWDROID}'s\ Memory\ Usage}{\text{MERGEDROID}'s\ Memory\ Usage}$ exhibits a strong positive relationship, with a Pearson correlation coefficient of 0.95.

To assess the memory impact of maintaining two additional data structures, *SymbolIncoming* and *Symb2Reps*, we measured their memory usage using Java instrumentation APIs. On average, as seen in Figure 5, these two data structures account for 7.1% of the total memory consumed during the analysis. Since MERGEDROID consistently uses less memory overall than FLOWDROID for all evaluated apps (Table I), the added overhead from maintaining these structures is negligible.

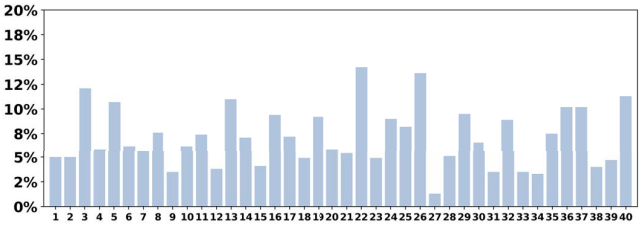


Fig. 5: Percentage of memory consumed by *SymbolIncoming* and *Symb2Reps* over the total memory of MERGEDROID for each app, identified by its ordinal number in Table I.

VI. RELATED WORK

Various approaches have been proposed to improve the efficiency of FLOWDROID [4], including sparse analysis [9], heap snapshot-assisted optimization [26], streaming-based parallelism enhancement [11], disk-assisted optimization [10], [27], and memory reclamation [8], [25]. However, MERGEDROID stands out by improving both efficiency and precision in FLOWDROID through the consolidation of equivalent value flows—a unique approach that differs from these existing ones.

While some approaches, like IccTA [5], focus on enhancing the precision of FLOWDROID by extending its intercomponent data leak detection capabilities, MERGEDROID refines FLOWDROID's precision by introducing context-sensitive activation statements to mitigate spurious value flows.

Numerous taint analysis tools, including Amandroid [28], DidFail [29], IccTA [5], DroidSafe [30], P/Taint [31], EvoTaint [32], CHEX [33], and DROIDINFER [34], have been proposed alongside FLOWDROID in recent years. However, FLOWDROID, rooted in the IFDS taint analysis framework, remains the most widely adopted tool in this domain.

The IFDS algorithm was initially introduced by Reps et al. [6] as a foundation for many critical applications, such as taint analysis [4], pointer analysis [35]–[37] and bug detection [38]. It was later generalized to the IDE algorithm [39] for solving inter-procedural distributed environment problems. As a popular algorithm, IFDS has been extended or improved by recent works [9], [40], [41] and has been implemented in many popular compiler frameworks, including WALA [14], SOOT [42], [43], and LLVM [44], [45].

VII. CONCLUSION

In this paper, we propose a novel *merge-and-replay* algorithm to enhance the performance of IFDS-based taint analysis. By consolidating equivalent value flows and integrating context information into activation statements, our approach reduces unnecessary propagation and prunes spurious value flows. We have implemented a prototype tool, MERGEDROID, and demonstrated that it significantly improves the efficiency and precision of FLOWDROID on a range of Android apps.

ACKNOWLEDGMENTS

We thank our reviewers for their constructive comments. This research is supported by an ARC grant DP210102409.

REFERENCES

[1] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Detecting cross-language memory management issues in Rust," in *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer, 2022, pp. 680–700.

[2] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. USA: USENIX Association, 2005, p. 18.

[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006, pp. 6 pp.–263.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps." in *PLDI*. ACM, 2014, pp. 259–269.

[5] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15. IEEE Press, 2015, p. 280–291.

[6] 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, pp. 49–61.

[7] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. New York, NY, USA: IEEE, 2015, pp. 426–436.

[8] 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)*. New York, NY, USA: IEEE, 2021, pp. 1098–1110.

[9] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: IEEE, 2019, pp. 267–279.

[10] 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)*. New York, NY, USA: IEEE, 2021, pp. 236–247.

[11] X. Wang, Z. Zuo, L. Bu, and J. Zhao, "DStream: A streaming-based highly parallel IFDS framework," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2488–2500.

[12] F-Droid. [Online]. Available: https://f-droid.org

[13] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical Extensions to the IFDS Algorithm," in *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144.

[14] WALA, "WALA: T.J. Watson Libraries for Analysis," 2023. [Online]. Available: http://wala.sourceforge.net/

[15] D. He, J. Lu, and J. Xue, "Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. Ali and J. Vitek, Eds., vol. 222. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:29.

[16] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 243–262.

[17] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, 1981, ch. 7, pp. 189–234.

[18] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, Citeseer, 1991.

[19] L. Li, C. Cifuentes, and N. Keynes, "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

[20] J. Lu, D. He, and J. Xue, "Selective context-sensitivity for k-CFA with CFL-reachability," in *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*, ser. Lecture Notes in Computer Science, C. Dragoi, S. Mukherjee, and K. S. Namjoshi, Eds., vol. 12913. Springer, 2021, pp. 261–285. [Online]. Available: https://doi.org/10.1007/978-3-030-88806-0_13

[21] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: Association for Computing Machinery, 2002, pp. 1–11.

[22] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 17–30. [Online]. Available: https://doi.org/10.1145/1926385.1926390

[23] S. S. E. G. at Paderborn University and F. IEM. (2023) DroidBench: an open test suite for evaluating the effectiveness of taint-analysis tools specifically for Android apps. [Online]. Available: https://github.com/secure-software-engineering/DroidBench

[24] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "TaintBench: Automatic real-world malware benchmarking of Android taint analyses," *Empirical Softw. Engg.*, vol. 27, no. 1, jan 2022.

[25] D. He, Y. Gui, Y. Gao, and J. Xue, "Reducing the memory footprint of IFDS-based data-flow analyses using fine-grained garbage collection," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023, p. 101–113.

[26] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller, "Heaps'n leaks: How heap snapshots improve Android taint analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1061–1072.

[27] 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*. New York, NY, USA: Association for Computing Machinery, 2017, p. 389–404.

[28] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1329–1341.

[29] 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*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–6.

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

[31] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[32] 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*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 265–269.

[33] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–240.

[34] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for Android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, Jul. 2015, pp. 106–117.

[35] J. Späth, L. N. Q. 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26.

[36] D. He, J. Lu, and J. Xue, "IFDS-based context debloating for object-sensitive pointer analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, jan 2023.

[37] ——, "Context debloating for object-sensitive pointer analysis," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: IEEE, 2021, pp. 79–91.

[38] 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*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 167–177.

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

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

[41] S. Arzt and E. Bodden, "Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes," in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 288–298.

[42] 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*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 3–8.

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

[44] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for C/C++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 393–410.

[45] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* New York, NY, USA: IEEE, 2004, pp. 75–86.