Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection

Steven Arzt
Secure Software Engineering Group,
Fraunhofer Institute for Secure Information Technology
Darmstadt, Germany
Email: steven.arzt@sit.fraunhofer.de

***Abstract*—Static data flow analysis is an integral building block for many applications, ranging from compile-time code optimization to security and privacy analysis. When assessing whether a mobile app is trustworthy, for example, analysts need to identify which of the user's personal data is sent to external parties such as the app developer or cloud providers. Since accessing and sending data is usually done via API calls, tracking the data flow between source and sink API is often the method of choice. Precise algorithms such as IFDS help reduce the number of false positives, but also introduce significant performance penalties. With its fixpoint iteration over the program's entire exploded supergraph, IFDS is particularly memory-intensive, consuming hundreds of megabytes or even several gigabytes for medium-sized apps.**

In this paper, we present a technique called CLEANDROID for reducing the memory footprint of a precise IFDS-based data flow analysis and demonstrate its effectiveness in the popular FlowDroid open-source data flow solver. CLEANDROID efficiently removes edges from the path edge table used for the IFDS fixpoint iteration without affecting termination. As we show on 600 real-world Android apps from the Google Play Store, CLEANDROID reduces the average per-app memory consumption by around 63% to 78%. At the same time, CLEANDROID speeds up the analysis by up to 66%.

I. INTRODUCTION

Static dataflow analysis is a well-researched area. Its algorithms build the foundation for many applications, especially in security and privacy analysis. Modern mobile applications process large amounts of sensitive personal data such as contacts, e-mails, banking and health data, text messages, or location data. Several stakeholders have incentives to gain access to such data, usually without the user's notice, be it for targeted advertising [1]–[3] or impersonation [4]. To prevent such unauthorized data leaks, mobile applications must be vetted for how they operate on sensitive data, i.e., which data they access, and where they transfer this data. Due to the large number of apps in modern app stores, manually inspecting each application before publishing it to end users is clearly infeasible. Therefore, automated techniques are required.

The properties in question are traditional data flow properties. Apps obtain sensitive data by calling API methods provided by the respective framework or operating system such as Android. If they need to transfer data to remote locations, e.g., via network connections, they again need to call an API method. The semantic transmission of the data record from the phone to the remote server directly translates to a code-level data flow between two API calls [5]. Especially when dealing with large amounts of applications on specialized platforms such as mobile devices [6]–[8] or large amounts of possible code configurations [9], static data flow analysis is a common choice. Dynamic analyses are often not a suitable alternative, because of the well-known code coverage problem [10]. Additionally, dynamic analyses on thousands of apps are infeasible on real-world mobile devices, despite recent advances in scalability [11]. Even though emulators exist, they require the complete mobile operating system to be emulated along with the app, which is significant overhead. Furthermore, the performance of emulating an ARM-based mobile phone on an x64 desktop computer or server is poor, and not all apps are available for the rather uncommon x86-based mobile OS versions [12], [13].

Static data flow analysis must be precise and avoid false positives to be suitable for a mass inspection of hundreds or even thousands of apps. Therefore, analyzers such as FlowDroid rely on the context-, and flow-sensitive IFDS framework [14], [15], and its implementations for popular program analysis frameworks [16]. The IFDS algorithm reduces the data flow problem to graph reachability in a structure called the *exploded supergraph*. The number of nodes in the exploded supergraph is in $O(n * m)$ where n is the number of statements in the program and m is the number of possible different data flow abstractions. Section II explains IFDS in more detail. This paper focuses on the memory consumption of an IFDS-based data flow analyzer, which is largely driven by the size of the exploded supergraph. Modern apps have hundreds of thousands of lines of code. If we assume that a taint abstraction encodes the tainted variable, there are hundreds of thousands of different taint abstractions in an analysis, leading to a huge exploded supergraph. In fact, as we show in our evaluation in section V, this amounts to hundreds of megabytes or even gigabytes for modern apps of moderate size. While such hardware requirements might still be feasible for a small number of apps on a dedicated machine, it effectively hinders

analyzing many apps at once, as it would be required for vetting a large app store within a realistic time frame.

Note that IFDS performs a fixed-point iteration on the exploded supergraph. Modern IFDS implementations such as Heros [16] or FlowDroid's FastSolver [17] reduce their memory footprint by incrementally building the exploded supergraph. A node in the graph for a statement `S` and a data flow abstraction `D` is only created once the taint `D` actually arrives at `S`, instead of building the maximum graph SxD upfront. Nevertheless, the resulting graphs are huge. Since building these graphs is essentially a fixpoint operation that terminates when no additional edges are added to the graph anymore, the graph grows continuously. Once an edge has been added to the graph, it remains in the graph until the analysis has finished. When creating the edge, the analysis cannot determine whether it will later encounter a loop or recursive call in the code and thus return to the same nodes in the exploded supergraph. It therefore assumes that each new edge is potentially relevant for detecting that a fixpoint has been reached and to ensure the termination of the analysis. Consequently, the memory consumption grows over time.

The key observation of this paper is twofold. Firstly, the majority of edges are only visited once, and are therefore not actually relevant for ensuring the termination of the analysis. Secondly, there is a *frontier* in taint propagation, i.e., the current set of edges over which taints are propagated. The IFDS solver can only return to edges that are reachable from this frontier. Inversely, all other edges can be removed. While propagation continues, the frontier shifts and more edges become redundant. This paper presents CLEANDROID, an approach for efficiently computing the frontier of the taint propagation as well as the set of redundant edges. It then removes the redundant edges from the exploded supergraph to regain memory, while guaranteeing that the overall data flow analysis still terminates. Note that removing edges also implicitly removes those taint abstractions that are no longer referenced by any edge, therefore freeing up significant amounts of memory. We consider our approach to be a semantic garbage collector for IFDS-based analyses.

Computing the frontier and identifying the redundant edges introduces additional computational overhead. We therefore evaluate the effects of different computation strategies (e.g., when to perform garbage collection) on memory consumption and computation time. Particular challenges arise in modern, multi-threaded IFDS solvers that propagate taints as one independent work item per edge. When new taints are created, each taint is propagated individually, potentially in different threads, even if all of them originate from the same statement. Consequently, the frontier is never stable, i.e., the set of taint abstractions at any given statement can change at any time depending on thread scheduling. There is no inherent ordering when taints arrive at a particular statement or when the taint set of that statement is complete. Garbage collection cannot introduce such synchronization points without massive adverse effects on the performance and scalability of the solver. We designed CLEANDROID such that it operates on a "moving target" without stopping the solver threads.

This paper presents the following original contributions:

- CLEANDROID, an approach for efficiently garbage-collecting edges from an IFDS exploded supergraph to regain memory during the analysis,
- an implementation of CLEANDROID in the FlowDroid open-source data flow analysis tool, and
- an evaluation on the performance (time and memory) effects of CLEANDROID in FlowDroid on 600 real-world Android apps from the Google Play Store.

We will contribute CLEANDROID to the FlowDroid open-source project once this paper has been accepted. Note that our approach is applicable to IFDS in general, and not limited to a particular implementation. The remainder of this paper is structured as follows: In section II, we give a short introduction into the IFDS framework, before presenting the CLEANDROID approach in section III. We discuss our implementation in section IV. In section V, we evaluate the time overhead and the memory gains of our approach, before presenting related work in section VI and concluding the paper in section VII.

II. BACKGROUND

This section explains the key concepts and algorithms to which this paper presents novel extensions.

A. IFDS

In their 1995 paper, Reps et al. [14] define a set of problems called IFDS (inter-procedural, finite, distributed, subset) and provide an algorithm for solving them. Many data flow problems fall in this category, such as taint tracking for privacy-sensitive data in Android apps [17] or checks for injection vulnerabilities [18]. The set of possible data flow abstractions is finite, if we assume the abstraction to simply reference the tainted variable (or access path [19], which is a variable followed by a sequence of field dereferences such as `var.f1.f2`[1]). The sets of tainted variables or access paths for two different paths through the inter-procedural control flow graph can be computed independently, and unioned at the join point, i.e., the statement in the code where the two paths meet. Therefore, the problem is a distributed subset problem.

The core idea of IFDS is to reduce the data flow problem to a graph reachability problem. There is one node for each possible combination of access path and statement, denoting that a specific access path is tainted at a specific statement. An edge from one node to another indicates that if the source of the edge holds (e.g., "x is tainted at statement S"), the destination of the edge (e.g., "y is tainted at statement T") also holds. Tautologies are placed at the sources ("z is tainted"), represented as edges from the special `zero` node to the respective access path.

The graph that consists of the pairs of statements and access paths as nodes and the effects of statements on taints as edges is called the *exploded supergraph*. To detect whether data

[1]Access paths are generally unbounded for recursive data structures, but techniques exist to approximate them to a finite subset [20].

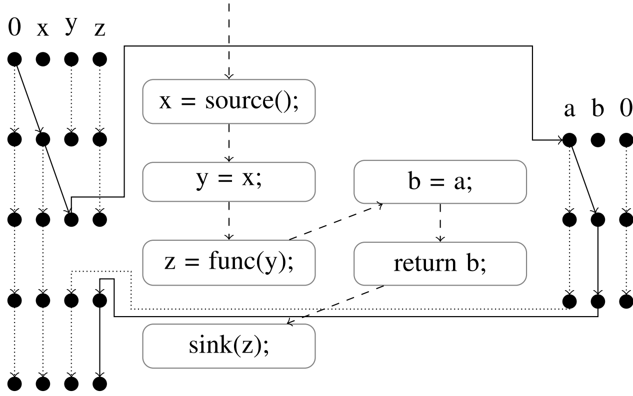

Fig. 1: Example for exploded supergraph

obtained from a source API call at a statement `S` is leaked through a sink API call at a statement `T`, the analyzer searches for a path from node `(0, S)` to node `(x, T)`, assuming that `x` is the variable that is passed into the sink. Figure 1 shows an example. The left side code portion is the main method with the program entry point. The right side code portion is a callee. The dotted arrows between the statements represent the interprocedural control flow graph. The solid black lines in the exploded supergraph indicate the path from the tautology `(0, x=source())` to the sink node `(z, sink(z))`. The dashed lines inside the exploded supergraph represent edges that are computed, but not relevant for the path in question.

Note that modern IFDS implementations [15], [16] do not compute complete exploded supergraph upfront and then perform a graph search. Instead, they incrementally extend the graph, i.e., start at each source `(0, S)` with the respective access path, and then incrementally add new edges by processing each transitive successor statement in the interprocedural control flow graph. The operation finishes once a fixpoint is reached and no further edges are added. This approach avoids adding unnecessary edges to the graph. In practice, only a minority of all possible combinations of statement and access path can actually receive a taint. Conversely, only a small subset of the entire exploded supergraph is actually required for the analysis. Furthermore, client analyses can check whether a sink has been reached when propagating new edges, i.e., *during* taint propagation. Therefore, they never perform a full graph search to find leaks, and thus only need the graph for their termination checks on the fixpoint iteration.

Still, even such minimal fractions of the exploded supergraph can grow very large as we show in section V-B. For ensuring that a fixpoint is reached when incrementally adding new edges, classical IFDS implementations never remove any edges. Note that the interprocedural control flow graph may contain loops, e.g., due to recursive method calls or loops in the program code. Therefore, keeping a set of edges that have already been traversed is essential for termination, i.e., for detecting that the fixpoint has been reached. In this paper, we present and evaluate techniques for *selectively* garbage-collecting edges in this set at *appropriate stages* during the overall fixpoint iteration.

Summaries are another key concept of IFDS. In the example, an incoming taint on the first parameter of the callee always leads to the return value being tainted, regardless of the concrete call site. The analysis therefore creates a context-specific summary. Whenever it encounters another call site for the same callee in which the first argument is tainted as well, it immediately taints the return value (i.e., applies the summary) without propagating any edges through the callee. We will use this property in section III to delete edges from callees that are guaranteed to be fully replaced by summaries.

Also note that modern multi-threaded IFDS solvers propagate their edges point-wise and independently. Each edge to be propagated is a new work item that is picked up by the next available worker thread. This allows the solver to use as many threads as there are edges currently being worked on, which effectively exploits modern multi-core processors. Therefore, even if one incoming taint at one statement yields a set of new abstractions to be propagated (i.e., new edges from the same start node in the exploded supergraph), these edges are all independent. There is no notion of a path being complete, aside from the overall fixpoint.

B. IDE

CLEANDROID is applicable to arbitrary IDE [21] problems as well. IDE is an extension to the IFDS framework and consists of two phases, where the first phase propagates path edges with functions as taint abstractions. In the second phase, IDE computes concrete values along the path edges in the exploded supergraph by computing the respective sequence of functions on a given input value. We observe that the first phase is equivalent to IFDS, only with a special universe of taint abstractions. Nothing in CLEANDROID is specific to a certain type of taint abstraction. CLEANDROID garbage-collects path edges in the first phase of IDE, which reduces the memory consumption for the exploded supergraph. Note that the second phase, which is by far less memory-intensive, remains unchanged. In fact, popular IFDS solvers such as Heros [16] are built on top of IDE solvers, and CLEANDROID would be integrated into the IDE solver in such a case.

C. IFDS-Based Alias Analysis

FlowDroid uses an IFDS-based alias analysis implemented through a second instance of an IFDS solver, which operates on an inverse (backward) interprocedural control flow. Whenever a heap object is tainted, the forward solver, which handles the normal taint propagation, injects an edge into the backward solver. The backward solver propagates this edge backward, since aliases are introduced *before* they are used. Once an alias is found, the backward solver injects a new taint on the respective access path into the forward solver, which propagates the taint on the alias along with all other “normal” taints. With passing edges between solvers, FlowDroid is able to retain context-sensitivity in the alias analysis. For the details of the two interleaved solvers, we refer the reader to the original paper [6]. In the context of this work, it is sufficient to note that edges may be propagated by *two different*

solvers, which each build up their own independent exploded supergraphs. This property is important for the notion of edge *liveliness*, which we introduce in section III. Note other IFDS-based works [18] also use multiple interleaved solvers, i.e., this property is not specific to FlowDroid. We therefore design CLEANDROID to support multiple solvers.

D. Callgraph Analysis

CLEANDROID requires an inter-procedural control flow graph (ICFG), which combines intra-procedural control flow with a callgraph in a single graph. We re-use the existing ICFG analysis in FlowDroid without any modifications, as FlowDroid precisely models the Android application lifecycle. It generates a dummy main method that serves as the program entry point for Soot's SPARK callgraph algorithm [22]. Since CLEANDROID uses the ICFG as a black-box input, it is not specific to FlowDroid. Note that loops and recursive calls are equivalent in the ICFG, since the ICFG contains call/return edges as well as edges inside methods.

III. APPROACH

The challenge of garbage-collecting edges in the exploded supergraph is equivalent to identifying those edges that will not be visited by the IFDS solver anymore. We call these edges *redundant edges*. We call edges that are not redundant edges, i.e., that can still be visited, *live edges*. When checking whether a given edge is a redundant edge or a live edge, this current edge is called a *candidate edge*. In other words, a candidate edge is live if the solver may generate a set of intermediate edges in the exploded supergraph *in the future* such that its propagation returns to the candidate edge. Falsely considering a candidate edge as live wastes memory and reduces the effectiveness of the garbage collector. Falsely considering a candidate edge as redundant, on the other hand, may lead to a non-terminating solver. Therefore, the algorithm should conservatively only consider those edges as redundant that are definitely unreachable from all edges that are currently being propagated. We call the edges that are currently being processed the *frontier* of the taint propagation. Note that, in a multi-threaded solver, the set of frontier edges is the union of the edges in the solver's worklist that are currently waiting for an executor with free capacity, and those edges that are currently being worked on. Frontier edges are, by definition, always live. Not all live edges are frontier edges, though. Recall that an edge is live when the solver *transitively* returns to it in the future, which may require starting at a frontier edge and then propagating an arbitrary number of intermediate edges before reaching the live edge in question.

As explained in section II-C, we design CLEANDROID for IFDS analyses with multiple solvers. We assume that each solver keeps its own exploded supergraph. Each solver can inject arbitrary edges into the other solver for propagation. We call a set of solvers that may inject edges into each other a *solver peer group*. From the perspective of a single solver, a new edge can appear at any node, without requiring any path to that node inside the current solver's exploded supergraph, as it may be contributed by another solver. We therefore define the union of all frontiers of all solvers inside a peer group as the *global frontier*. There may be multiple peer groups, but the global frontier is always relative to a single peer group. Similarly, a *globally redundant* edge is redundant for all solvers in a peer group. If an edge is only redundant for a subset of solvers, we call it *locally redundant*. As an example of dependencies between solvers, let us assume that we remove an edge that is only locally redundant for solver S but not for solver T in the same peer group from the exploded supergraph of S. In this case, solver T may inject the same edge back into S. Then, assume the same edge becomes locally redundant for T and is removed from T's exploded supergraph, until S re-injects the edge into T, which re-creates the original situation. Due to the intermittent removals followed by re-injections, none of the two supergraphs reaches a fixpoint and the analysis never terminates. We therefore conclude that only globally redundant edges may be removed from any exploded supergraph in any solver of the peer group. To simplify the presentation, we assume the global properties unless explicitly stated otherwise or obvious from context (e.g., *frontier* usually refers to *global frontier*).

Each solver runs its own garbage collector. The garbage collectors of all solvers in a peer group form a *garbage collector peer group*. We use the term *peer group* interchangeably for solver peer groups and garbage collector peer groups in the respective context. Whenever a solver schedules a new edge for propagation, this new edge becomes part of the frontier and the solver registers it as live with its associated garbage collector. When an edge has been processed (i.e., the graph has been extended with potentially new edges beyond the current one), it is no longer part of the frontier, and unregistered from the garbage collector. This allows each garbage collector to keep track of the local frontier for its corresponding solver.

Listing 1: Target Program for IFDS Analysis

```
1  public void main(String[] args) {
2    int a = source();
3    for (int i = 0; i < 100; i++) {
4      int b = callee1(a);
5      a = callee2(b);
6    }
7  }
```

We now use the example in listing 1 to give a better intuition of live edges in an interprocedural analysis. We later use the same example to explain an efficient approximation for live edges. For simplicity, we describe a traditional Java program instead of an Android app. In the example, we consider the methods `callee1` and `callee2` to be arbitrary computations of significant code size, whose details are not important. As explained in section II, the IFDS analysis only terminates when reaching a fixpoint on the edges of the exploded supergraph. In the example, we assume that the global frontier only contains edges from method `callee1`. We observe that the analysis will later return from `callee1`,

call callee2, and return from callee2 again. It will then taint variable a in the main method before following the control flow graph back to the loop header, and finally arriving back at the call to callee1. Therefore, the edge that brings variable a into the loop is live, as it is visited transitively from the frontier in method callee1. An ideal garbage collector would *precisely* compute the set of all outgoing paths through the exploded supergraph from the frontier and mark all edges on these paths as live. In the example, this would include all edges in the loop to ensure that the incremental generation of this cyclic subgraph reaches its fixpoint. However, the problem of precisely enumerating the live edges (and thus checking whether a candidate edge is live or redundant) is equivalent to the IFDS problem. Edges are created incrementally and the full exploded supergraph is only known when the full IFDS problem has been solved. While creating this graph, the analysis cannot already traverse the edges that are yet to be created, to see whether it will re-visit a certain candidate edge in the future.

To avoid this problem, CLEANDROID applies conservative over-approximation of live edges. Our approach reduces the frontier computation problem to a graph search in the immutable call graph of the program, which is not only much smaller than the exploded supergraph, but which can be pre-computed before the IFDS analysis begins. We observe that the solver can only re-visit the edges in a particular method, if it can return to that method via a control flow edge. In the following, we build a recursive over-approximated definition of live edges around this observation. Note that considering too many edges live reduces the effectiveness of the garbage collection, but does not impact the correctness or termination property of the IFDS algorithm. In fact, CLEANDROID gracefully degenerates to basic IFDS in the worst scenario. Therefore, the approach retains these properties of IFDS, as long as the approximation is complete, i.e, contains at least those methods in which the solver may propagate edges in the future. Reducing the computation to a callgraph search is efficient, because since plain IFDS already requires the interprocedural control flow graph, which is in turn based on the callgraph. Therefore, this data structure already exists in a plain IFDS analysis and does not require additional work to implement CLEANDROID. In practice, we translate the callgraph into a map from each method to the set of transitive callees of that method for faster queries. The time and memory required to create and store this map, respectively, are not significant. As we show in section V, CLEANDROID reduces both the time and memory consumption of the IFDS analysis in comparison to the original FlowDroid implementation.

Our approximation is recursively defined as follows. We consider a method as live if there is at least one edge known to be live in that method. Recall that frontier edges are live, i.e., they are the base case in our definition. A method is also live if it (directly or transitively) calls a method that is already known to be live. All edges inside a live method are considered live as well. A method that is not live is called *redundant*. All edges inside a redundant method are considered redundant. The basic idea behind this approximation is as follows: We assume that all methods in which the solver is currently propagating edges or to which the solver might return (i.e., transitive callers) may potentially contain loops. All these potential loops may lead to cyclic subgraphs in the exploded supergraph, i.e., we conservatively consider all edges in these methods live. CLEANDROID makes no attempt to verify whether the solver may actually return to a specific edge inside any of these methods (over-approximation).

We use the example in listing 1 to explain why our approximation of live methods is complete, i.e., no live methods are considered redundant by our approximation. Note that redundant methods may be considered live, though. We also show why our definition of live methods only needs to consider method returns and not method calls. In the example, we again assume a state in which all current frontier edges belong to method callee1. Therefore, method callee1 is live. It might contain a loop, and removing edges from inside a loop may lead to non-termination. More precisely: removing edges from this part of the exploded supergraph therefore poses the risk that this edge is a part of a cyclic subgraph, and removing it would lead to an intraprocedural infinite propagate/remove cycle. Method main is live, because its callee callee1 is live. In other words, the IFDS solver eventually returns to main from callee1 via a return edge. Removing edges from main can therefore lead to the issue we described when introducing the example. If the taint abstraction on variable a is always removed while working on callee1, and re-created upon return, before processing callee1 again, an infinite propagate/remove cycle may occur.

Method callee2, on the other hand, is not live. It does not contain any frontier edges, nor does it call any live method. We now explain why this does not impact termination. Recall from section II that IFDS creates method summaries for methods that have already been processed. When the solver processes a method call for the same callee and context for which a summary exists, it applies the summary instead of propagating edges through the callee again. The edges in methods for which summaries exist can therefore safely be removed. Consider method callee2 in the example. If a summary for a given incoming taint abstraction already exists and the call is encountered once again, the solver applies the summary instead of propagating edges through callee2. If no summary exists for the current context (i.e., the incoming taint abstraction), the solver propagates through the method *once*. While this propagation is running, the method is live, because its edges are on the frontier and thus live. Alternatively the solver has descended into transitive callees, which also makes the method live, since it has live callees. Once the solver has completed analyzing the transitive callees rooted in callee2, and the method is no longer live, the summary exists. Therefore, the edges inside callee2 are never visited again for this context and can be deleted safely. Note that edges are context-sensitive just like summaries. Therefore, CLEANDROID only deletes those edges that correspond to the same context for which the summary was created. CLEANDROID

never deletes summary edges. In the notation of Naeem et al. [15], we only remove entries from the `PathEdge` table.

Since CLEANDROID is designed for IFDS problems with multiple interacting solvers, each garbage collector queries its entire peer group whether a particular method is redundant. The edges inside a method are only removed if the method (and thus all edges in it) are globally redundant in all solvers of the peer group. Note that all solvers share the same control flow graph, allowing for a shared cache of transitive callee relationships already queried before.

Algorithm 1 shows the implementation of the garbage collector. The global variable λ maps methods to the number of unfinished edge propagations in that method, while variable `c` contains the candidate methods for garbage collection once they are no longer live. Method `Initialize` is called before the IFDS solver schedules its first edge for processing. Method `RunGarbageCollector` is run in a separate thread concurrently to the garbage collector and constantly checks for edges that can be removed. The IFDS solver invokes method `OnEdgeScheduled` whenever it schedules a new edge for processing. Note that we define the method to take a full edge $\langle d_1, n, d_2 \rangle$ as parameter to be consistent with common presentations of the IFDS algorithm [15]. Once an edge has been taken from the worklist and has been fully processed, the solver calls method `OnEdgePropagated`. The IFDS solver only needs to be modified to call these methods in the garbage collector and can otherwise remain unmodified. Frontier computation is handled in method `TransitiveCallees`, which obtains the set of transitive callees of a given method via a callgraph search. With a call to `MethodOf`, the garbage collector can retrieve the method that contains a given statement. In the algorithm, we iterate over all edges in the exploded supergraph (represented by the edge set `Graph`) to find all edges in a given method and remove them one by one. In our actual implementation, we instead adapted the solver to quickly remove all edges within a given method without having to iterate over all edges in the graph.

Method `RunGarbageCollector` is run concurrently to the other methods. It computes the set of live methods based on a method reference counter, which is concurrently being modified by `OnEdgeScheduled` and `OnEdgePropagated`. When the solver processes an edge that leads to one or more new edges being added to the worklist, it calls `OnEdgeScheduled` on all of them. Only then, it marks its current edge as done via `OnEdgePropagated`. Therefore, CLEANDROID works on an over-approximation of live edges, to prevent infinite sequences of edge removal and re-creation as explained above. As we show in section V-D, even on a highly-concurrent system with 144 cores and as many IFDS worker threads, CLEANDROID is thread-safe.

Recall that FlowDroid checks for leaks while propagating edges, i.e., whenever it processes a statement, it checks whether this statement is a sink and whether the incoming data flow abstraction references a variable passed to that sink. Therefore, removing edges from the exploded supergraph has no effect on the correctness of the data flow analysis, as long as all sink edges are propagated at least once.

Algorithm 1: Garbage Collection Algorithm

GLOBAL VARIABLES: λ – method reference counter, c – candidate methods

Function `Initialize()`:
1. $\lambda = (x \mapsto 0)$
2. $c = \emptyset$

Function `RunGarbageCollector()`:
3. **while** `AnalysisIsRunning()` **do**
4.   **foreach** $m \in c$ **do**
5.   **if** $\lambda(m) = 0$ **then**
6.   **if** $\forall m' \in$ `TransitiveCallees`(m) : $\lambda(m') = 0$ **then**
7.   `// Remove the edges for method` m
9.   **foreach** $e : \langle d_1, n, d_2 \rangle \in Graph$ **do**
11.   **if** `MethodOf`$(n) = m$ **then**
13.   `RemoveEdgeFromSolver`(e)
15. $c = c \setminus \{m\}$

Function `OnEdgeScheduled(`d_1, n, d_2`)`:

INPUT: d_1 – incoming abstraction, n – current statement, d_2 – outgoing abstraction
OUTPUT: nothing

16. $m =$ `MethodOf`(n)
17. $\lambda = \begin{cases} x \mapsto \lambda(x) + 1 & \text{if } x = m \\ x \mapsto \lambda(x) & otherwise \end{cases}$
18. $c = c \cup \{m\}$

Function `OnEdgePropagated(`d_1, n, d_2`)`:

INPUT: d_1 – incoming abstraction, n – current statement, d_2 – outgoing abstraction
OUTPUT: nothing

19. m = `MethodOf` (n)
20. $\lambda = \begin{cases} x \mapsto \lambda(x) - 1 & \text{if } x = m \\ x \mapsto \lambda(x) & otherwise \end{cases}$

IV. IMPLEMENTATION

We implemented CLEANDROID as an extension to the FlowDroid open-source data flow solver with about 400 lines of code. FlowDroid's architecture supports integrating new solvers that can be selected by the configuration. Our new *garbage collecting solver* is an adapted version of FlowDroid's original FastSolver IFDS solver. A garbage collector run is a sequence in which the garbage collector first computes the redundant edges on its associated solver. It then limits this set to those edges that are also globally redundant. This subset is then removed from the solver's exploded supergraph. We

TABLE I: Category Distribution of the Data Set

Category	Apps	Category	Apps
Age Range 1	22	Game Music	3
Age Range 2	27	Game Puzzle	18
Age Range 3	27	Game Racing	9
Android Wear	23	Game Role Playing	20
Art and Design	7	Game Simulation	16
Auto and Vehicles	6	Game Sports	6
AR Core	7	Game Strategy	12
Books and Reference	6	Game Trivia	1
Business	4	Game Word	10
Communication	16	Health and Fitness	9
Dating	17	House and Home	3
Education	2	Libraries and Demos	2
Entertainment	9	Lifestyle	15
Events	5	Maps and Navigation	5
Family Action	8	Medical	17
Family Braingames	3	Music and Audio	11
Family Create	3	News and Magazines	18
Family Education	9	Parenting	2
Family Music Video	7	Personalization	9
Family Pretend	5	Photography	16
Finance	3	Productivity	15
Food and Drink	2	Shopping	10
Game Action	10	Social	12
Game Adventure	14	Sports	8
Game Arcade	19	Tools	15
Game Board	11	Travel & Local	18
Game Card	9	Video Players	11
Game Casino	3	VR Device	1
Game Casual	12	Weather	6
Game Educational	6		

do not share multiple solvers (and thus multiple exploded supergraphs) between a single garbage collector, because the load (the number of new edges per time unit) can be largely different between solvers, and we want to allow for potentially different garbage collector configurations per solver in the future. Each garbage collector (one per solver) runs in a separate thread and regularly performs runs while the IFDS solver is running. After each run, it pauses for the *GC interval*, to allow the IFDS solver to extend the exploded supergraph before searching for the next set of redundant edges.

V. Evaluation

In this section, we evaluate CleanDroid with regard to the following research questions:

- RQ1 How much time and memory does FlowDroid consume on our data set (baseline)?
- RQ2 How does CleanDroid affect the memory consumption of FlowDroid?
- RQ3 How does CleanDroid affect the time consumption of FlowDroid?
- RQ4 How does CleanDroid perform on large apps?
- RQ5 How does CleanDroid affect the correctness of FlowDroid?

A. Experimental Setup

The experiments described in this section were conducted on a randomly-selected set of 600 free apps from the Google Play Store, downloaded in spring 2020. Table I shows the apps' category distribution. If an app is in multiple categories, we list the first and usually most relevant category.

TABLE II: Machine Configurations for the Evaluation

	Machine M1	Mach. M2	Mach. M3
CPU Cores	144	4	8
CPU Type	Intel Xeon Gold 6154	AMD EPYC 7542	
Physical Memory	3 TB	64 GB	
Java Heap Size	250 GB	50 GB	
Machine Type	Physical	Virtual	

The focus of this paper is on the relative difference in memory and time consumption between CleanDroid and the base case. We used three different machines for the evaluation as shown in table II. All machines run the Oracle JDK version 1.8.0 on Ubuntu Linux 18.04. Our high-performance machine M1 allows us to test the effect of CleanDroid in the case of a highly parallel data flow analysis. Recall that there are two solvers with one garbage collector thread each to clean up data flow facts. On the other hand, there are 144 threads (one per core as per FlowDroid's default settings) that create new data flow facts, putting maximum pressure on the garbage collector. Since the solvers' worklists do not always contain 144 items, the solver does not put full load on the system. Consequently, the CleanDroid garbage collector threads usually do not compete with the solver threads for CPU cores on M1. On M2 and M3, on the other hand, with far fewer cores, such competition occurs frequently.

For calculating the required memory, we did not rely on FlowDroid's default metric, which queries the JVM for its total amount of memory, from which it subtracts the amount of free memory reported by the JVM. We found this method to be unreliable, e.g., reporting a memory consumption in the tens of megabytes, even with millions of abstractions present in the exploded supergraph. External memory analysis tools consequently also reported much larger values, sometimes in the gigabytes. However, these tools were not immediately applicable to our analysis as they only capture snapshots for a later offline analysis. These snapshots are as large as the consumed memory and costly to write to disk. Since the exact contents of the objects are irrelevant to our experiments and we only require precise sizing, we instead integrated Twitter's memory calculation library for Java [2] into our evaluation tool. With this library, we measured the actual size of the exploded supergraph (represented by the path edge table and all objects reachable through it) in both the forward and backward solver of FlowDroid. Note that we excluded the size of the Soot data objects that represent the app's parsed classes, methods, statements, etc., since they are loaded regardless of the concrete analysis and are not specific to FlowDroid or the data flow problem.

For the FlowDroid base case without CleanDroid, we measured the memory consumption once per solver after the data flow analysis has finished. Since no edges are ever removed, this method is accurate. For CleanDroid, we measured the memory consumption every 15 seconds in a separate thread. This approach reduces the negative effect of the sampling on the performance of the IFDS solver. Otherwise, we would risk propagating significantly fewer edges

[2]https://mvnrepository.com/artifact/com.twitter.common/objectsize

TABLE III: Baseline Performance Results (All Apps)

Metric	Avg.	Median	Min.	Max.
Taint Propagation Time	152s	80s	1s	307s
Memory Forward Solver	3,369 MB	2,809 MB	0 MB	12,259 MB
Memory Backward Solver	1,557 MB	1,220 MB	0 MB	12,589 MB
Total Solver Memory	4,926 MB	4,191 MB	0 MB	16,539 MB
Abstractions Forward	1,152	829	2	7,342
Abstractions Backward	652	367	0	5,716
Total Abstractions	1,805	1,208	2	13,058
Sources	81	51	2	680
Sinks	439	360	2	2,032
Classes	17,441	14,890	57	103,216
Methods	100,761	85,720	173	546,653
Statements	188,103	146,466	1,500	892,514

TABLE IV: Baseline Performance Results (No Timeout)

Metric	Avg.	Median	Min.	Max.
Taint Propagation Time	14s	2s	1s	82s
Memory Forward Solver	484 MB	13 MB	0 MB	3,723 MB
Memory Backward Solver	195 MB	4 MB	0 MB	1,413 MB
Total Solver Memory	679 MB	17 MB	0 MB	5,135 MB
Abstractions Forward	516	168	2	3,271
Abstractions Backward	275	72	0	1,778
Total Abstractions	791	240	2	5,049
Sources	44	25	2	313
Sinks	311	226	3	1,101
Classes	14,135	13,901	238	51,366
Methods	84,252	79,614	1,249	332,869
Statements	115,602	80,946	1,500	467,512

than in the base case before the timeout is triggered. The final per-app memory consumption is the maximum over all samples and the final size of the edge cache after the analysis has completed. Still, we measured memory consumption and required time in separate runs for CLEANDROID for further reducing the impact of the measurement overhead on the results. To get consistent snapshots, the memory analysis needs to intermittently pause other threats (i.e., lock on the individual data objects being measured). We wanted to ensure that these (albeit minimal) effects do not affect our timing evaluation. Note that the final size of the exploded supergraph need not be zero when running with garbage collection enabled. Edges may be propagated between the last run of the garbage collector and the end of the analysis.

We ran FlowDroid's callback collection phase once per app, serialized the results, and re-used it for all data flow runs to ensure that all runs work on the same callgraph. We used a timeout of five minutes on the taint propagation part of the data flow analysis. For the path reconstruction, we did not specify a timeout. Selecting appropriate sources and sinks is an orthogonal area of work [5]. This paper focuses on the delta of time and memory consumption between FlowDroid's IFDS solver with and without CLEANDROID's garbage collection when both are used with the same set of sources and sinks. We therefore used an adapted version of FlowDroid's default source/sink definition file[3].

B. *RQ1: Baseline Experiment*

In this section, we report on the baseline of the FlowDroid data flow solver on our dataset without CLEANDROID. On 12 apps, the analysis did not complete, because either the APKs were invalid, or due to errors in the analysis. Table III shows our measurements for the remaining 588 apps, on machine M1. On 80 apps (about 14%), the analysis timed out. Note that the average timings over all apps are inaccurate, because they are always bound by roughly the 5 minute cut-off [4]. To avoid this issue, table IV shows the values only for the remaining 508 apps that FlowDroid processed without a timeout.

We observe that FlowDroid consumes significant amounts of memory during the data flow analysis. While up to 16 GB are still acceptable for analyzing a single app, it does not scale to mass analyses. App store providers, for example, need to check all uploaded apps for security vulnerabilities and privacy violations [23]. Since stores contain millions of apps, many of which are regularly updated [12], they must scan dozens or even hundreds of apps concurrently. CLEANDROID is an important step towards making precise approaches such as FlowDroid applicable to such use cases.

We repeated our baseline evaluation on machines M2 and M3. As expected, the analyses require significantly more time on M2 and M3 than on the high-performance compute server M1. For the apps without timeout, FlowDroid takes 33s on M2 on average and 31s on M3. This is about 2.4 times, and 2.2 times the values for M1, respectively. The timings for all apps, including those with timeouts, are biased by the 300s timeout. We therefore find only a 37% increase for both M2 and M3. The increased analysis time did, however, not increase the number of timeouts, which is 75 apps (about 13%) on M2 and 74 apps on M3. Note that slight variations in timeouts are expected for apps with timings close to the timeout. These apps may time out in one run, and complete in another one with just seconds of difference in timing.

We found 166 apps (28%) to require more than 1 GB of memory. On these apps, FlowDroid requires 8.3 GB of memory and takes 260s on average on machine M1. All of the 80 apps on which FlowDroid times out require more than 1 GB of memory, i.e., the 166 high-memory apps are a superset of the 80 apps with timeouts. On only the remaining 86 large apps without timeout, FlowDroid requires 3.1 GB of memory and takes 60s for the analysis on average. We found the analysis time to be almost equal on all three machines. If we exclude the large apps over 1 GB, the average memory consumption of the remaining apps is 18 MB.

C. *RQ2: Memory Evaluation of* CLEANDROID

Figure 2 shows a comparison between the average baseline memory consumption (solid line) and CLEANDROID's average memory consumption for different GC intervals on machine M1. We ran each app with CLEANDROID, configured with the respective interval (x value). We then measured the average and median memory consumption of all apps, and recorded it as the y value in the graph. Running with longer intervals ($x > 10$) did not provide any new insights. The graph on the left side shows the data for the entire data set. The graph on

[3]We removed some overly broad source definitions to reduce the number of timeouts in the base case.

[4]Recall that FlowDroid's timeouts are not totally precise, since they prevent new IFDS edges from being processed, but allow all solver threads to finish their current work item, i.e., edge.

the right side includes only those apps on which the data flow analysis did not time out. The lines represent the baseline, solid blue for the average memory consumption, dashed red for the median memory consumption over the respective app set (all apps or only those without timeout). The black bullets represent the averages over the apps with CLEANDROID for the respective GC interval. The red diamonds show medians.

We observe that CLEANDROID significantly decreases FlowDroid's memory consumption in all cases. For all apps, the memory gain is between 11% ($x = 9$) and 63% ($x = 0$) on average. For the apps on which the data flow analysis did not time out, the average memory gain is between 22% ($x = 1$) and 78% ($x = 0$). Continuously running the garbage collector, i.e., not pausing between garbage collector runs ($x = 0$), offers the highest memory gain in both cases. For the apps on which FlowDroid timed out, CLEANDROID provides a memory gain of up to 23%. In the worst case, CLEANDROID degenerates to the base case. When considering the median memory consumption, the reduction on the entire app set is almost 100%. For $x = 0$, the median baseline memory consumption for all apps is 3,297 MB, whereas CLEANDROID only requires about 5 MB.

For $x > 0$, the memory gain is less than for $x = 0$. This is expected, because the garbage collector pauses, i.e., allows more edges to accumulate in the IFDS solver before a new garbage collection is triggered. More edges in the solver require more memory. However, this process is non-deterministic because of the non- deterministic thread scheduling between the 144 worker threads of the IFDS solver and the two worker threads (one per solver) for the garbage collector. Therefore, increasing the intervals does not steadily lead to higher memory consumption.

We measure similar memory gains on machines M2 and M3. On machine M2, the memory consumption is reduced by 68% for all apps, and by 59% for the apps that did not time out. On machine M3, the memory gain is 70%, and 87%, respectively. All values were measured for $x = 0$.

D. RQ3: Time Evaluation of CLEANDROID

Figure 3 compares the required data flow time for the base case (solid line) with the time required in CLEANDROID on machine M1. The presentation is similar to the memory evaluation presented before. The graph on the left side shows the timings for all apps in our data set. The graph on the right side shows the timings of only those apps on which the data flow analysis completed without a timeout. Since the number of apps with timeouts is equal for the base case and CLEANDROID, the timeout of 300 seconds does not influence the validity of the averages presented in the figure, although it introduces an upper limit for the timings. We focus on the effect of CLEANDROID on the time and memory consumption of FlowDroid. Therefore, we only consider relative differences between the base case and CLEANDROID, not the absolute number of seconds or timeouts.

Although CLEANDROID was not designed for increased speed, we observe that our approach actually *reduces* the time required for the data flow analysis. Investigation with a profiling tool showed that deleting edges from the IFDS solver reduces the load factor of the hash maps that store the edges. Consequently, fewer re-hashings were necessary to insert new edges afterwards, because the maps had more free entries left in their backing data structures. For all apps, CLEANDROID is between 5% ($n = 9$) and 66% ($n = 0$) faster than the baseline on average. The median speedup is 99% - from 80s to one second for $x = 0$ on the entire data set. For the apps for which the data flow analysis completed without a timeout, CLEANDROID is between 0% ($n = 7$) and 57% ($n = 0$) faster than the baseline on average. The median timings are almost identical to the baseline for all apps.

On machine M2, the average data flow analysis is 46% faster (113s instead of 208s) for all apps. For the subset of apps without timeout, the data flow time decreased by 61% on average (13s instead of 33s). On machine 3, the analysis for all apps is 35% faster (136s instead of 209s). For the apps without timeouts, the speedup is 38% (18s instead of 25s). All timings were measured for $x = 0$.

E. RQ4: Evaluation of CLEANDROID on Large Apps

For all apps with a memory consumption of more than 1 GB in the baseline, including those with a timeout, CLEANDROID achieves an average reduction by 25% on machine M1, by 54% on M2, and by 53% on M3. In the median, the reduction is 27% on M1, 49% on M2, and 49% on M3. For the apps that were analyzed without a timeout, CLEANDROID saves 31% on M1, 46% on M2, and 54% on M3 on average (median: 33% on M1, 44% on M2, and 55% on M3). Note that the lower percentages on the large apps are outweighed by the larger absolute numbers, since these apps require 8.3 GB of memory on average in the base case. Therefore, CLEANDROID still provides considerable savings. The timings were identical between the base case and CLEANDROID (within a 5% margin) for the large apps, i.e., CLEANDROID degenerates to the base case with respect to timings on large apps.

F. RQ5: Correctness Evaluation of CLEANDROID

We compared the leaks detected by the FlowDroid baseline with the results of CLEANDROID for the 508 apps on which both FlowDroid and CLEANDROID terminate without a timeout on machine M1. We found that CLEANDROID had no influence on the leaks discovered by FlowDroid, i.e., identified the same set of leaks for each app.

G. Discussion

We observe that CLEANDROID performs best when run continuously, i.e., without pausing between garbage collector runs ($x = 0$). This is expected, because continuous cleanup reduces the number of abstractions that accumulate in the IFDS solvers. CLEANDROID reduces the memory consumption by between 63% (all apps) and 78% (apps without timeout). It speeds up the analysis by between 66% and 57% respectively. The effect of longer pauses between GC cycles ($x = 1, 2, 3...$) is non-deterministic, and depend on the operating system's

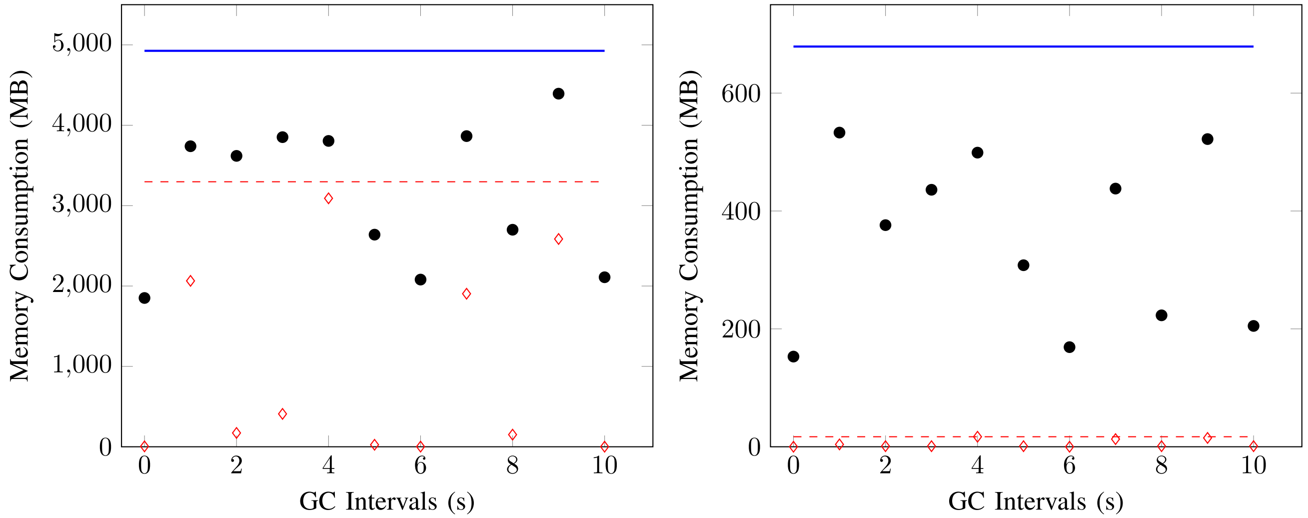

Fig. 2: Average Memory Consumption over GC Intervals (left: all apps, right: apps that did not time out, blue solid line: baseline average, red dashed line: baseline median, black bullets: CLEANDROID average over the apps, red diamonds: CLEANDROID medians over the apps)

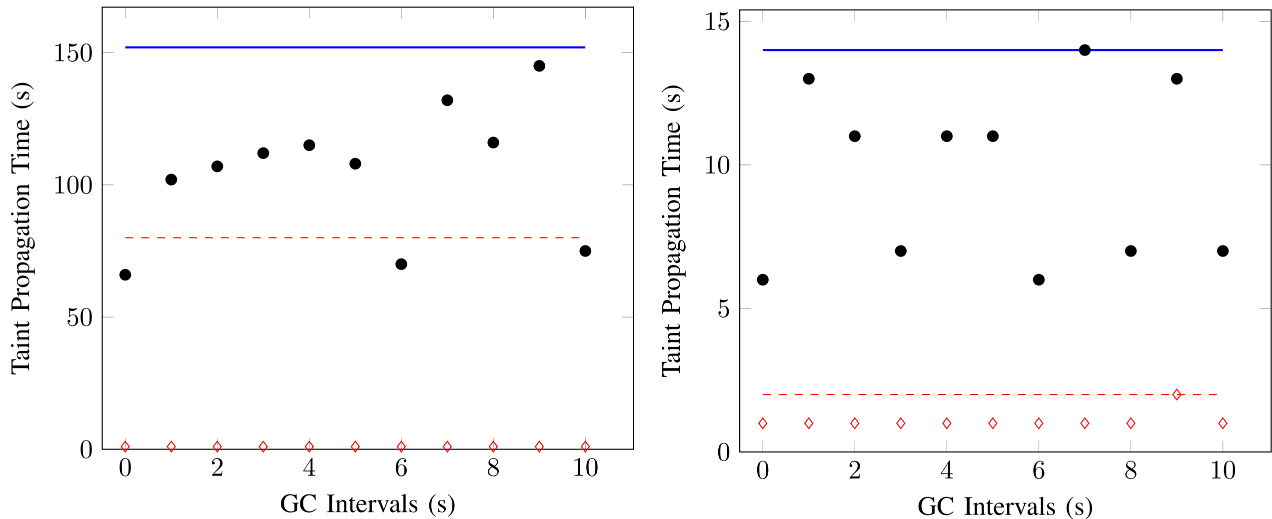

Fig. 3: Taint Propagation Time over GC Intervals (left: all apps, right: apps that did not time out, blue solid line: baseline average, red dashed line: baseline median, black bullets: CLEANDROID average over the apps, red diamonds: CLEANDROID medians over the apps)

thread scheduling and the precise properties of each app (e.g., average out-degree of a node within the exploded supergraph).

We find that CLEANDROID also performes well on fewer cores, although FlowDroid's worker threads and the CLEANDROID cleanup threads compete for cores in such as scenario, potentially leading to fewer garbage collector runs. For large apps, CLEANDROID can save gigabytes of memory (around 50% of 8.3 GB on average), making large-scale analyses on app stores more efficient.

H. Threats to Validity

Machine M1 is physical hardware, while M2 and M3 are virtual machines. The shared host machine for M2 and M3 uses different hardware than M1. Therefore, the timing values for M1 and M2/M3 might not be directly comparable to the values for M1. We find a larger standard deviation in the timings on the VMs already in the base case, most likely due to VM scheduling and varying loads on the host machine.

VI. RELATED WORK

Various researchers have improved the performance of data flow analyses. While reducing the solver's memory consumption is usually not the prime goal, approaches that reduce the number of propagated edges (usually for performance improvement) also implicitly reduce the memory consumption. None of these approaches, however, solve the problem that the exploded supergraph can only grow, and never shrink.

Bodden [24] argues that carefully designing the abstract domain of the taint abstractions can benefit precision as well as performance. Lerch et al. [25] provide guidance on designing such abstractions. Access-Path Abstraction [26] is a technique for generalizing access paths to reduce the overall number of different taint abstractions that are propagated through a program. All of these works focus on improving summary re-use in IFDS and thus a smaller exploded supergraph and less memory consumption. He et al. [27] propagate data flow facts not along the control flow graph, but directly to the their respective next possible use point. This approach skips all statements in between and does not create edges for them. Weiss et al. [28] use a database as a fallback for IFDS problems with high memory consumption. They store the exploded supergraph in a hybrid disk/memory configuration, effectively using disks when memory is exhausted. Graspan [29] uses big data graph analysis based on disks. These approaches are othogonal to CLEANDROID. An ideal version of FlowDroid would incorporate all of these approaches along with CLEANDROID to make abstractions as small as possible, remove them from memory as early as possible (CLEANDROID's contribution), and write them to disk if memory is still exhausted.

Other approaches avoid re-computing the same data flows multiple times. Reviser [30] attempts to incrementally update the data flow results after changes to the code under analysis, e.g., after each build in continuous integration. This approach reduces the overall time consumption, but does not affect the peak memory requirement. StubDroid [31] pre-computes library summaries for later re-use during app analysis. StubDroid is orthogonal work and already part of FlowDroid and therefore of the base case in this work. Both the base case and CLEANDROID use StubDroid summaries. Rountev et al. [32] present similar work for IDE analyses [21], an extension to IFDS. While Rountev et al.'s work summarizes IDE / IFDS edges, StubDroid summarizes the effects of the library method on access paths. In either approach, the exploded supergraphs of all libraries, which would normally be subgraphs of the analysis on the programs that use these libraries, are replaced by summary edges. Consequently, these approaches inherently reduce the memory consumption of the analysis.

Aside from taint tracking, researchers have also worked on symbolic execution for data flow analysis, including mobile targets such as Android [33]. Others have extended Java-based abstract interpretation frameworks to run precise data flow analysis on Android [34], or have focused on model checking for data flow properties [35], [36]. Taint tracking and symbolic execution can achieve the same degree of completeness, and neither approach offers superior performance in general [37]. In an attempt to combine the different approaches, work has been done to post-process taint tracking results using symbolic execution [38]. Such a combined approach would benefit from CLEANDROID in its taint tracking phase. Kim et al. [39] have improved the memory efficiency of abstract interpretation [40] (AI), which is more generic than IFDS. While data flow analysis has been implemented in AI [41], IFDS can exploit the distributivity property for improved baseline efficiency. Comparing CLEANDROID to an equivalent AI-based data flow tracker with Kim et al.'s optimization is future work.

Weighted Pushdown Systems (PDS) [42] and Synchronized Pushdown Systems (SPDS) [43], [44] are alternative methods for precise static data flow analysis. They can efficiently encode recursive access paths without loss of precision. We plan to apply CLEANDROID to the automaton's saturation process `post*` in Späth's work as future work. In this algorithm, we can define a frontier on the nodes and edges that are incrementally added, and can check whether the algorithm can return to a certain node that has been added previously. If not, the node can be removed. This will yield an incomplete automaton, similar to CleanDroid, which yields an incomplete exploded supergraph. Still, all SPDS data flow analyses can be refactored to check for sinks inside of `post*`, i.e., while building the automaton, instead of querying the final automaton. The automaton is only required for termination, which is not affected by CLEANDROID's garbage collection.

Garbage collection for heap objects has been a prominent research topic, especially for managed programming languages such as Java. Classic stop-the-world collectors stop all worker threads to identify objects that are no longer reachable. Modern approaches, on the other hand, try to run mostly in parallel, but still require a sequential phase in which all worker threads are paused [45]–[48]. CLEANDROID does not require any sequential phase. Further, our approach must reason about an edge being re-visited *in the future*, i.e., via edges that do not yet exist at the time of garbage collection. In traditional garbage collectors, all references already exist when determining reachability.

VII. CONCLUSION

In this paper, we have presented CLEANDROID, an approach for efficiently garbage-collecting edges and taint abstractions inside an IFDS data flow solver. We have integrated our approach into the FlowDroid open source data flow solver, and evaluated it on 600 real-world apps. We have shown that CLEANDROID reduces FlowDroid's average memory consumption by up to 63% (entire dataset), and 78% (apps on which no timeout occurred). Further, CLEANDROID speeds up the data flow analysis by up to 66% and 57% respectively. In total, CLEANDROID is an important step towards scaling the highly-precise FlowDroid analysis to large numbers of apps, e.g., when checking all newly-uploaded apps in an app store for security and privacy issues. We further plan to optimize the implementation of CLEANDROID before contributing our work to the FlowDroid open source project.

ACKNOWLEDGMENT

This research work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity.

REFERENCES

[1] S. Zimmeck, J. S. Li, H. Kim, S. M. Bellovin, and T. Jebara, "A privacy analysis of cross-device tracking," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1391–1408.

[2] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," 01 2018.

[3] D. Malandrino, A. Petta, V. Scarano, L. Serra, R. Spinelli, and B. Krishnamurthy, "Privacy awareness about information leakage: Who knows what about me?" 11 2013, pp. 279–284.

[4] T. Nagunwa, "Behind identity theft and fraud in cyberspace: the current landscape of phishing vectors," *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, vol. 3, no. 1, pp. 72–83, 2014.

[5] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[6] 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," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[7] F. Wei, S. Roy, and X. Ou, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1329–1341.

[8] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 576–587.

[9] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba, "Spllift-transparent and efficient reuse of ifds-based static program analyses for software product lines," in *Proc. of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 355–364.

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[11] M. Miltenberger, J. Gerding, J. Guthmann, and S. Arzt, "Dfarm: massive-scaling dynamic android app analysis on real hardware," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, 2020, pp. 12–15.

[12] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM international conference on Measurement and modeling of computer systems*, 2014, pp. 221–233.

[13] S. Rasthofer, S. Arzt, M. Kolhagen, B. Pfretzschner, S. Huber, E. Bodden, and P. Richter, "Droidsearch: A tool for scaling android app triage to real-world app stores," in *2015 Science and Information Conference (SAI)*. IEEE, 2015, pp. 247–256.

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

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

[16] 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.

[17] S. Arzt, "Static data flow analysis for android applications," Ph.D. dissertation, Technische Universität Darmstadt, 2017.

[18] 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.

[19] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 210–225.

[20] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," *ACM Sigplan Notices*, vol. 29, no. 6, pp. 230–241, 1994.

[21] 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

[22] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Compiler Construction*, ser. Lecture Notes in Computer Science, G. Hedin, Ed. Springer Berlin Heidelberg, 2003, vol. 2622, pp. 153–169. [Online]. Available: http://dx.doi.org/10.1007/3-540-36579-6_12

[23] N. J. Percoco and S. Schulte, "Adventures in bouncerland," *Black Hat USA*, vol. 95, p. 110, 2012.

[24] E. Bodden, "The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them)," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 85–93. [Online]. Available: https://doi.org/10.1145/3236454.3236500

[25] J. Lerch and B. Hermann, "Design your analysis: A case study on implementation reusability of data-flow functions," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 26–30. [Online]. Available: https://doi.org/10.1145/2771284.2771289

[26] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 619–629.

[27] 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)*, 2019, pp. 267–279.

[28] C. Weiss, C. Rubio-González, and B. Liblit, "Database-backed program analysis for scalable error propagation," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 586–597.

[29] 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," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 389–404, 2017.

[30] 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*, 2014, pp. 288–298.

[31] ——, "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. Rountev, M. Sharp, and G. Xu, "Ide dataflow analysis in the presence of large object-oriented libraries," in *Compiler Construction*, L. Hendren, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 53–68.

[33] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson, "Checking interaction-based declassification policies for android using symbolic execution," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 520–538.

[34] Étienne Payet and F. Spoto, "Static analysis of android programs," *Information and Software Technology*, vol. 54, no. 11, pp. 1192 – 1201, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584912001012

[35] D. A. Schmidt, "Data flow analysis is model checking of abstract interpretations," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 38–48.

[36] J. Posegga and H. Vogt, "Java bytecode verification using model checking," in *Workshop Fundamental Underpinnings of Java*. Citeseer, 1998.

[37] M. Belyaev, N. Shimchik, V. Ignatyev, and A. Belevantsev, "Comparative analysis of two approaches to static taint analysis," *Programming and Computer Software*, vol. 44, no. 6, pp. 459–466, 2018.

[38] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, 2015, pp. 1–6.

[39] S. K. Kim, A. J. Venet, and A. V. Thakur, "Memory-efficient fixpoint computation," in *International Static Analysis Symposium*. Springer, 2020, pp. 35–64.

[40] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of

fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[41] M. Zanioli, "Information flow analysis by abstract interpretation," Ph.D. dissertation, Università Ca'Foscari Venezia, 2012.

[42] T. Reps, S. Schwoon, and S. Jha, "Weighted pushdown systems and their application to interprocedural dataflow analysis," in *International Static Analysis Symposium*. Springer, 2003, pp. 189–213.

[43] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

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

[45] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith, "A comparative evaluation of parallel garbage collector implementations," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2001, pp. 177–192.

[46] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer, "Implementing an on-the-fly garbage collector for java," *ACM SIGPLAN Notices*, vol. 36, no. 1, pp. 155–166, 2000.

[47] T. Printezis and D. Detlefs, "A generational mostly-concurrent garbage collector," *SIGPLAN Not.*, vol. 36, no. 1, p. 143–154, Oct. 2000. [Online]. Available: https://doi.org/10.1145/362426.362480

[48] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank, "A parallel, incremental, mostly concurrent garbage collector for servers," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, p. 1097–1146, Nov. 2005. [Online]. Available: https://doi.org/10.1145/1108970.1108972