An Efficient Android App Debloating Approach Based on Multi-layer Dependence Graph

Hengqin Yang[12], Jiwei Yan[13]✉, Jun Yan[123]✉, Bin Liang[4], Jian Zhang[12]
[1]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
[2]Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4]Renmin University of China, Beijing, China
{yanghq, yanjun, zj}@ios.ac.cn, yanjiwei@otcaix.iscas.ac.cn, liangb@ruc.edu.cn

***Abstract*—Android apps are getting bloated by continuously integrating possibly unnecessary functional modules. This trend of software bloat negatively impacts the performance of static analysis tools. As a result, analysis reports are more likely to contain false positives and experience analysis timeouts. Consequently, developers are forced to manually inspect and troubleshoot errors, as well as restart the analysis process, making analyzers more time-consuming and less user-friendly. However, existing approaches for Android app debloating almost only consider how to remove redundant code elements or functional features from the perspective of users, thus they are unsuitable for the analyzer-oriented app debloating task in most cases.**

To fill this gap, we propose an Android app debloating approach that employs a novel Multi-layer Dependence Graph (MDG) structure to represent the app under analysis. We hierarchically construct the MDG by sequentially analyzing and capturing dependence at the class, method, and statement levels. Throughout this process, we dynamically identify hotspot classes and narrow down the scope for further dependence extraction, thereby alleviating the challenge of a too complicated graph structure caused by the excessive app size. We implement our approach as the tool FlowSlicer, a novel MDG-based static Android app debloater. We evaluate FlowSlicer by utilizing it to debloat the input app first and then observing the performance difference of two analysis processes which accept the original and the debloated app as input respectively. The evaluation is performed on both the hand-crafted and the real-world apps in our benchmark. Our results show that FlowSlicer is not only capable of effectively debloating Android apps but also enhancing the performance of static analyzers. For instance, cooperating with FlowSlicer, the analyzer FlowDroid could detect 212 more leaks in real-world apps in our benchmark.

***Index Terms*—Android, Software Debloating, Static Analysis**

I. Introduction

Android currently dominates the global mobile operating system market, holding a 71.95% market share in the first quarter of 2023 [20]. Owing to the portability and convenience of mobile devices, an increasing number of mobile apps have emerged within the Android ecosystem, becoming indispensable to everyday life [21]. To ensure the quality and security of Android apps, numerous static analyzers have been developed and tailored for features specific to the Android framework. These analyzers play a crucial role for developers in identifying code vulnerabilities, ensuring code quality, and enhancing security through automated checks. For instance, one of the renowned static taint analyzers for Android is FlowDroid [6], which introduces a novel approach by constructing a dummyMain class as the entry point for its analysis. This technique allows for accurate integration of Android component lifecycle methods into the call graph, enabling precise modeling of taint data flow and effective detection of potential data leaks in Android apps.

Behind the prosperity of the Android market lies a largely overlooked phenomenon—an escalating trend known as "software bloat", which is rapidly spreading across the entire Android ecosystem. Software bloat, defined as the results of continuously adding new features to software programs to the point where the benefit is outweighed by the defects of resources consuming [17], [24], is gradually becoming a common phenomenon for Android apps. For instance, as the largest standalone mobile app in China, WeChat's installation package size has increased nearly 600 times in eleven years from 457KB in January 2011 (version 1.0) [1] to 264MB in August 2024 (version 8.0.50) [4], which significantly depicts the current situation of overall Android app size bloating faced by users and developers.

As for the negative efects of software bloat, a bloated app may raise various privacy, safety, and resource consumption problems [17]. From a developer's perspective, performing static analysis on bloated apps often leads to inefficiencies, as static analyzers may get trapped in infinite loops due to the excessive code volume and the complexity of integrated functionalities. This, in turn, prevents the tools from producing comprehensive analysis results. For example, static taint analysis aims to track the propagation of sensitive data within a program. In the context of Android apps, a data leak occurs when private information (e.g., phone numbers, device identifiers) flows from sensitive sources to public sinks (e.g., the Internet, SMS transmission), resulting in unintended exposure. Automated taint analyzers, such as FlowDroid [6], are designed to trace tainted information throughout an app and provide precise insights into how data leaks occur. However, prior research has demonstrated that most static taint analyzers struggle to yield reliable results when analyzing real-world

✉ Corresponding authors.

apps collected from Google Play [38]. Therefore, it is crucial to propose efficient and effective approaches to mitigating the phenomenon of software bloat in Android apps.

To alleviate Android app bloat, both industry and academia have proposed a series of debloating approaches and developed tools to assist users in removing code elements of the target app. Google Play has set a maximum size limit for app bundles, feature modules, and asset packs that users can download from the platform [12]. It also released a series of static analyzers to help developers understand the APK structure, inspect the source code, and reduce app size [11]. Prior academic studies also have presented several approaches to debloating Android apps. Jiang et al. [17] considered the dead code as bloated code and identified the dead code in Android apps statically. Tang et al. [30] identified user-defined features from different perspectives statically, and asked developers to debloat the app by only retaining the selective features. To sum up, existing tools primarily focus on reducing the size of Android apps downloaded by end users. However, none of them have implemented a debloating approach specifically tailored for the static analysis process, i.e., these approaches are largely ineffective in addressing the challenges developers encounter when using static analyzers to analyze bloated apps.

To fill this gap, we propose FlowSlicer, a tool that automatically debloats Android apps through multi-layer dependence graph (MDG) construction and program chopping technology. Specifically, we define and extract the class-, method- and statement-level dependence hierarchically through static analysis techniques. First, we perform reachability propagation at the class level to identify hotspot classes while excluding irrelevant ones from further analysis, thereby effectively controlling the size of the constructed dependence graph. Then we merge method- and statement-level dependence to handle inter-procedural behaviors and construct the MDG as the program representation. By solving the vertex reachability problem on the MDG, we obtain a pruned, refined MDG as the result of program chopping. Based on this chopped MDG, we eliminate the corresponding code elements, ultimately achieving debloating of the input app.

We have conducted a series of experiments to evaluate the effectiveness, efficiency, and code removal capability of our approach on both hand-crafted apps and complex real-world apps. We utilize FlowSlicer to debloat apps before inputting them into the analyzer FlowDroid. The evaluation results show that FlowSlicer is capable of not only retaining true positive leaks but also enhancing the precision of FlowDroid through eliminating false positives. On hand-crafted apps, our tool successfully identifies all original TP leaks and reduced 38.09% FPs compared to the original results; it helps reduce the total FlowDroid time of real-world apps by 67.29% and achieves an average 59.91% FlowDroid time reduction after debloating. Cooperating with FlowSlicer, FlowDroid could detect 56 new leaks on the non-time-out benchmark and 156 new leaks on the time-out benchmark of real-world apps. Moreover, FlowSlicer is capable of removing unnecessary statements significantly on all our benchmarks.

In summary, this work makes the following contributions.

- We give the definition of the multi-layer dependence graph (MDG) and achieve a lightweight dependence analysis via constructing the MDG.
- We propose an Android app debloating approach tailored for static analyzers based on the MDG.
- We implement our debloating approach in a publicly available tool FlowSlicer. The evaluations on hand-crafted and real-world apps show that FlowSlicer is not only capable of effectively debloating Android apps but also enhancing the performance of static analyzers.

II. Preliminaries and Motivating Example

This section introduces the related preliminary knowledge and presents a motivating example to illustrate our ideas.

A. Preliminaries

1) Taint Analysis: Taint analyzers aim at tracking sensitive "tainted" information by starting at a pre-defined source and then following the data flow until it reaches a given sink, reporting precise information about how the sensitive data may be leaked. Taint analysis can be implemented both statically and dynamically. We mainly focus on static taint analyzers for Android, i.e., those that track taint propagation by analyzing Android app code without actually running it, including FlowDroid [6], Amandroid [33], DroidSafe [13], etc. These tools primarily differ in their design choices aimed at balancing precision and scalability in the analysis process.

2) Program Slicing and Chopping: Since its introduction in the 1980s [14], [35], program slicing has been employed for numerous tasks on programs including testing, debugging, analysis, understanding, etc. A slice of a program is taken concerning a program point p and a variable x; the slice consists of all statements of the program that might affect the value of x at point p. The tuple $\langle p, x \rangle$ is defined as the corresponding slicing criterion. Program chopping [15], [16] is a generalization of program slicing, which expands the concept of a slicing criterion into two sets of variable instances called source and sink. Chopping then yields the sub-program that shows how the definitions of the source instances can affect the uses of the sink instances.

3) Dependence Graph: Slicing and chopping are related operations, and both can be performed by solving reachability problems in a dependence-graph representation of the program [14], [15], [28]. In the case of intra-procedural slicing/chopping, the slice/chop can be obtained by solving a reachability problem on the procedure's dependence graph. To solve the inter-procedural slicing/chopping problem, whereas the dependence graphs for all procedures need to be collected together to form a global dependence graph. The results of traversing the global dependence graph are equivalently mapped to the results of inter-procedural slicing/chopping.

B. Motivating Example

In this subsection, we present an example to illustrate how our approach is motivated. Considering the alignment

```
1
2  public class MainActivity extends Activity {
3    ...
4    protected void onCreate(Bundle bundle) {
5      super.onCreate(bundle);
6      setContentView(R.layout.activity_main);
7    }
8
9    private boolean isOkToShow() {
10     PackageManager packageManager =
           this.ctx.getPackageManager();
11     List l = packageManager.
           getInstalledApplications(128);//Source
12     Collection c = checkForLaunchIntent(l);
13     for (ApplicationInfo appInfo : c) {
14       String str2 = packageManager.getPackageInfo
15       (appInfo.packageName, 0).versionName;
16       //Leak1: source -> appInfo -> sink
17       Log.d("package name",
               appInfo.packageName);//Sink
18       //Leak2: source -> appInfo -> str2 -> sink
19       Log.d("package version name", str2);//Sink
20     }...}}
21
```

Listing 1. A Motivating Example with Two Leaks.

between the taint sources and sinks identified by taint analyzers and the sets of source and sink variables required for program chopping, we design our Android app debloating approach specifically to support tainted data-flow leak detection tasks. The code snippet in Listing 1 shows the taint leaks in a real-world Android app. From Listing 1 we can see that there are two obvious leaks in the MainActivity class, which both flow from the source API `getInstalledApplications(int)` to the sink API `d(java.lang.String, java.lang.String)`. The concrete leak paths are all marked in red.

To comprehensively detect potential data leaks before executing the app, one way is to adopt widely used static taint analyzers such as Flowdroid. However, in our experiments using FlowDroid's `runInfoflow()` API with default configurations, no taint flow leaks were reported within the predefined timeout period. To investigate this issue, we manually examined the analysis reports and traced FlowDroid's workflow step by step. Finally, we observed that the analysis was significantly delayed during the data-flow solving phase, where the framework attempts to determine whether any feasible path exists from a source to a sink.

This performance bottleneck primarily stems from the complexity of solving data-flow equations over a large domain of data facts. Specifically, FlowDroid adopts the IFDS framework [27], whose time complexity is $\mathcal{O}(|E| \cdot |D|^3)$, where $|E|$ denotes the number of control-flow edges and $|D|$ denotes the size of the data-flow domain. Given this cubic dependency, the sheer number of data facts significantly impacts analysis performance and scalability.

Rather than directly applying taint analyzers, an alternative approach is leveraging Android app debloating tools to preprocess the app before conducting taint analysis. XDebloat, presented by Tang et al. [30], is a novel Android app debloating tool that allows users to customize the features they want to keep or remove in different granularity. However, the tool does not perform specific source and sink identification for taint analysis. It cannot effectively eliminate unnecessary code while preserving essential data-flow paths, which is a common limitation to all Android debloating tools targeted at end users. As for program slicers, we experimented with Jicer [25], a multi-functional static slicer for Android apps. However, applying Jicer to our motivating app presented several intractable challenges. For instance, the GUI may freeze when selecting slicing criteria, and Jicer may fail to execute properly due to the excessive size of the globally constructed dependence graph.

Motivated by these attempts, we propose a static analysis pre-processing approach that aims to reduce the size of the data-flow domain by eliminating code elements and corresponding data facts that are irrelevant to taint propagation. By pruning such irrelevant information prior to taint analysis, our apporach effectively reduces the computational burden of the IFDS solver, enabling faster and more scalable taint analyses.

III. Methodology

This section presents the workflow of our debloating approach and the details in each step.

A. Overview

Fig. 1 gives the overview of our debloating approach. The input is the original binary Android app, and the output is the generated app which has been debloated. As shown in Fig. 1, our approach processes the input app through two main steps. First, it performs the multi-layer dependence graph construction step (Section III-B) to build the dependence graph, which contains the intra- and inter-procedural dependence between statements in the target app. Then in the debloating stage (Section III-C), our approach solves the reachability problem on the dependence graph to form a chopped dependence graph and removes the unnecessary code elements of the app. To ensure the analyzability of the debloated app, it also replenishes necessary extra code (e.g., entry point callbacks and the corresponding statements in the Android app).

B. Multi-layer Dependence Graph Construction

Before the chopping stage, we first perform static analysis on the input app to extract the dependence through the level of class, method, and statement hierarchically. Specifically, we define five kinds of dependence relationships between classes, two kinds of dependence relationships between methods, and seven kinds of dependence relationships between statements in total. At the stage of collecting class-level dependence, we locate the source-sink invocations and execute the reachability propagation step to find out hotspot classes which will be reserved in the following method- and statement-level analyses. After extracting the method- and statement-level dependence, we will combine them to form a *Multi-layer Dependency Graph* (MDG) to serve the program chopping phase.

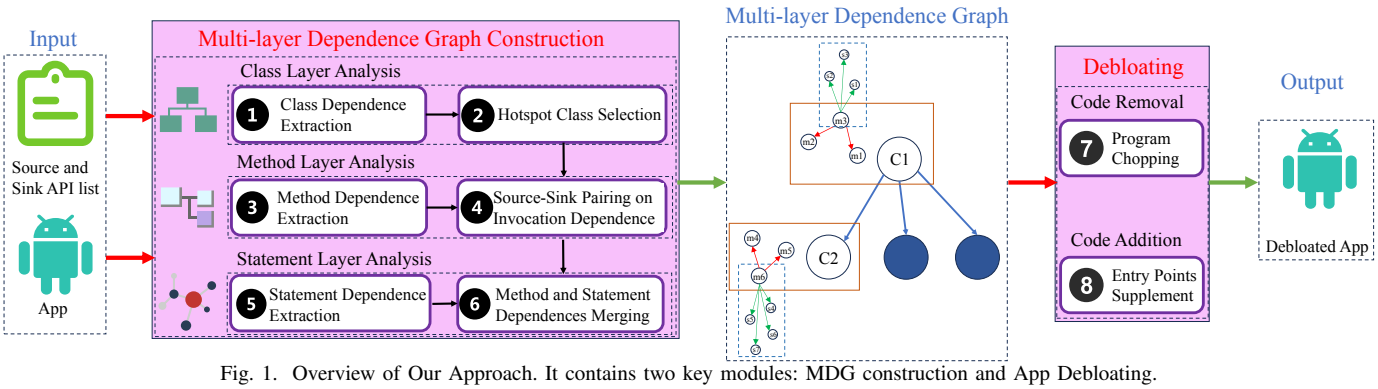

Fig. 1. Overview of Our Approach. It contains two key modules: MDG construction and App Debloating.

Since our MDG is different from the classic dependence graphs described in prior program slicing work [14], [28], [35] in the feature of hierarchical dependence extraction, we first define a *Complete Dependence Graph* (CDG) constructed from the input app and then present the difference between our MDG and the CDG in detail.

Given an input Android app $\mathcal{P}$, the definition of a complete dependence graph is as follows:

Definition 1. *The **complete dependence graph (CDG)** for app $\mathcal{P}$ is a directed graph $G = \langle B, V, R, E \rangle$, where*

- *B represents the basic information of the target program, which is a triple $\langle S, M, C \rangle$. Sets S, M, and C denote the sets of all statements, methods, and classes in $\mathcal{P}$.*
- *V is a set of nodes. Each node $v \in V$ is triple $\langle s, m, c \rangle$, which represents a statement $s \in S$ in the method $m \in M$ and the class $c \in C$.*
- *R is a set of statement-level dependence relationships. For each $r \in R$, r is statement-level dependence, which is a homogeneous relation over the set V, $r \subseteq V \times V$.*
- *E is a set of directed edges, which satisfies $E \subseteq V \times V$.*

In a CDG, the node set V comprises all statements from the input app, resulting in an excessively large dependence graph and significantly slowing down the subsequent debloating process. In contrast, when constructing our MDG, we first extract the class dependence and perform a reachability propagation step to retain only hotspot classes. By excluding irrelevant classes early in the process, we eliminate numerous unnecessary dependence extraction operations in later stages of the analysis. This substantially reduces the computational overhead and mitigates the challenges associated with analyzing a large-scale dependence graph.

We then introduce how to extract dependence in the order of classes, methods, and statements.

*1) **Class Level Analysis**:* Our multi-layer graph construction approach starts with the class dependence extraction step. The class-level dependence contains the class hierarchy relations, invocation relations, field dependence relations, etc. Given the input Android app P and the set C containing all classes in $\mathcal{P}$, the class-level dependence is defined as follows:

Definition 2. *The **class-level dependence** denoted by R_C is a homogeneous relation over the class set C, which contains*

- *the **class inheritance relation**: For $A, B \in C$, class A is a subclass of class B.*
- *the **interface implementation relation**: For $A, B \in C$, class A implements the interface B.*
- *the **outer class relation**. For $A, B \in C$, class A is the outer class of class B.*
- *the **field dependence relation**. For $A, B \in C$, an instance of class A is the field variable of class B.*
- *the **calling relation**. For $A, B \in C$, there exists a method m in class A that invokes a method n in class B.*

Class Dependence Extraction. We leverage the class hierarchy query APIs provided by the Soot Framework (e.g. the `getSuperclass()` method in `SootClass.java`) to capture the dependence between classes, including the inheritance relation, the interface implementation relation, and the outer class relation. In addition, if an instance of a class serves as a field variable of another class in a specific situation, the field dependence between the two classes will be collected by our approach. Lastly, we implement the Class Hierarchy Analysis (CHA) algorithm [10] to extract calling relationships between classes in cases where method invocations occur.

Hotspot Class Selection. After collecting the class-level dependence, we select the hotspot classes through transitive dependence relationships to narrow down the analysis scope. Specifically, in this step, we identify the source and sink API invocation sites in $\mathcal{P}$ and perform a reachability propagation to filter and exclude classes not reachable from original source-sink invocation sites through the dependence edges, thus obtaining the hotspot class set denoted by C_R. We use the denotation "R" to represent "reachability" and each class in the set C_R is reachable from or to the source-sink invocations through the class-level dependence.

Algorithm 1 depicts the complete process of source-sink identification and reachability propagation. As shown in phase 1 of algorithm 1, for a given app, our approach identifies and records its source-sink API invocation sites as the slicing criteria for the chopping step. FlowDroid provides a comprehensive and well-maintained list of default configurations for common source-sink APIs in Android app. By comparing with the API list in lines 5-13, we systematically detect all source-sink invocations appearing in the app, analyzing statements sequentially. Notably, since a few source and sink APIs listed

Algorithm 1 Source-Sink Identification and Reachability Propagation

Input: $\mathcal{P}$ - a given app.
$\mathcal{C}$ - the set of all classes in the app $\mathcal{P}$.
$\mathcal{L}$ - the default source and sink API list.
R_C - the collected class-level dependence.
Output: C_R - the hotspot classes.
{**Phase1: Source-Sink Identification.**}
1: $manifest \leftarrow processManifest(\mathcal{P})$;
2: $permissions \leftarrow readPermissions(manifest)$;
3: $sourceSinkMap \leftarrow \emptyset$;
4: $M \leftarrow \emptyset$;
5: **foreach** $c \in \mathcal{C}$, $m \in c.methods()$, $u \in m.units()$ **do**
6: **foreach** $api \in \mathcal{L}$ **do**
7: **if** $needsPermission(api)$ **then**
8: $(c, m, u) \leftarrow checkEquality(u, api, permissions)$;
9: **else**
10: $(c, m, u) \leftarrow checkEquality(u, api)$;
11: **end if**
12: **end for**
13: **end for**
{**Phase2: Reachability Propagation.**}
14: $graph \leftarrow buildAdjacencyGraph(R_C)$;
15: $sccGraph \leftarrow extractSCC(graph)$;
16: $L_S \leftarrow retrieveDetectedSourceSinkList()$;
17: $C_R \leftarrow DFS(L_S, sccGraph)$;
18: $sccGraphReversed \leftarrow reverseEdges(sccGraph)$;
19: $C_R \leftarrow DFS(L_S, sccGraphReversed)$;
20: **return** C_R;

by FlowDroid require user-granted permissions to be invoked normally, we collect the permissions defined in the manifest file of the app in line 2 and refer to them when judging whether a statement invokes the source-sink API or not in lines 7-10. Extracting permissions helps our approach avoid false positives in scenarios where there is a source-sink API invocation but no permission is granted.

Phase 2 of algorithm 1 gives the basic procedures of the reachability propagation. In line 14, we first construct the adjacency graph through class-level dependence and then extract the Strongly Connected Components (SCC) [31] to condense the adjacency graph into a Directed Acyclic Graph (DAG) in line 15. After that, we perform the depth-first search algorithm on the DAG in line 17 to record all traversed classes reachable from identified source-sink invocations. At the same time, we reverse the DAG through its edges and perform an identical depth-first search on the reversed graph in line 19 to collect classes reachable to the source-sink invocations. Finally, we obtain the set of hotspot classes as the output of algorithm 1, denoted by C_R. Thus we have managed to narrow the scope of analysis from the set $\mathcal{C}$ containing all classes in the app to the set C_R containing all reachable classes through class-level dependence.

*2) **Method Level Analysis**:* Method-level dependence primarily captures inter-procedural behaviors, including method invocations and data flows between fields and local variables. Given that parameter transmission and return values can propagate sensitive data, accurately modeling method invocation behaviors is both valuable and necessary for precise program analysis. However, the Android framework introduces significant challenges to invocation dependence extraction, leading to limited effectiveness in traditional approaches. One major difficulty arises from Android's event-driven programming model, which relies heavily on callback methods. These callbacks can be triggered by user interactions or system events, creating dynamic execution paths that are difficult to detect through static analysis. Additionally, Android components (e.g., Activities) define predefined lifecycle methods, which the framework invokes at various execution stages. Since the Android runtime manages the order of execution dynamically, accurately capturing invocation dependence across different lifecycle phases further complicates the analysis.

Given the set C_R and the set M_R containing all methods of C_R, we define the method-level dependence as follows:

Definition 3. *The **method-level dependence** denoted by R_M is a homogeneous relation over the set M_R, which contains*

- *the **method invocation relation**: For $A, B \in M_R$, method A invokes method B explicitly or implicitly.*
- *the **inter-procedural field-data relation**. For $A, B \in M_R$, method A defines the field variable used in method B.*

Method Dependence Extraction. In detail, we extract the invocation dependence from the following two aspects:

- **Callback Invocation.** We implement our invocation dependence extraction approach atop FlowDroid's dummyMain mechanism. FlowDroid provides interfaces for collecting lifecycle methods and callback methods, allowing us to systematically link them to the constructed dummyMain class in a flexible order. By leveraging dummyMain as a centralized entry point, we can traverse the call graph more efficiently, eliminating the need to process all potential entry points individually.
- **Implicit Method Invocation.** Android apps also allow implicit method calls. For reflection, to identify the target of the reflection, we utilize FlowDroid to identify all propagated string constants. For inter-component communication (i.e., ICC), we support inputting the analysis result of ICCBot [37] to detect ICC in the target app. For asynchronous tasks, we follow Tang et al. [30] and add the following edges to the call graph: $AsyncTask.execute() \rightarrow onPreExecute()$, and $onPreExecute() \rightarrow doInBackground()$, etc.

The inter-procedural field-data dependence between fields and local variables is modeled through a reaching definition analysis. Fields are never referenced directly in Jimple, they are always assigned to local variables before being defined or used. In consequence, to detect which definitions reach which uses of a certain field, we implement the classic reaching definition analysis approach [3] to conclude the field dependence relations through the def-use chains of fields in classes of the set C_R.

Source-Sink Pairing on Invocation Dependence. After extracting the invocation dependence and locating source-sink invocations in the given app, our approach performs a source-sink pairing step on the method-level dependence to figure out whether there are no invocations from a certain source API to

all sink APIs. Once this case happens, we will set these source APIs excluded from the following analysis, i.e., we will not take these API invocation statements as slicing criteria in the phase of chopping. Similarly, if there are not any invocations to a certain sink API, these sink APIs will not contribute to the leak detection and thus need to be excluded.

We implement the source-sink pairing approach through the state-of-the-art reachability querying algorithm named *IP+* [34], which utilizes a randomized labeling approach to compute labels and answer reachability queries on large-scale graphs. After constructing the adjacency graph according to the collected invocation dependence, we have successfully transformed the pairing problem on the method-level dependence to the reachability querying problem from source to sink API on the graph and utilized the *IP+* algorithm to update the detected source-sink invocations.

*3) **Statement Level Analysis**:* Given the method set M_R, we denote the set containing all statements of M_R by the set S_R and define the statement-level dependence to illustrate the dependence relationship between statements as follows:

Definition 4. *The **statement-level dependence** denoted by R_S is a homogeneous relation over set S_R, which contains*

- *the **control dependence relation**. For $A, B \in S_R$, the statement A is control-dependent on the statement B.*
- *the **data dependence relation**. For $A, B \in S_R$, the statement A is data-dependent on the statement B.*

Statement Dependence Extraction. In this step, we extract intra-procedural dependence between statements.

When a statement $s_j \in S_R$ is control-dependent on another statement $s_i \in S_R$, the execution of s_j depends on the execution of s_i. For example, s_i could be the head of an if-branch while s_j belongs to its body. We have manually added an entry node for every method and set the *return* statements and the *throw* statements as the exit nodes. The entry node has control dependence on all statements in the method.

A statement $s_l \in S_R$ is data-dependent on another statement $s_k \in S_R$, if a variable defined in s_k is used in s_l and there are no intervening uses in between. As for the implementation, we employ Soot's post-domination computation to finish the control dependence analysis and extract data dependencies between statements through an intra-procedural reaching definition analysis [3]. The data dependence analysis leverages StubDroid's [5] method summaries for Android and Java libraries, mapping inputs (e.g., method base, parameters) to outputs (e.g., fields, return values). This enables determining redefinitions of base objects or parameters without directly analyzing the library method, thus enhancing the scalability.

Method and Statement Dependencies Merging. To accomplish the inter-procedural analysis tasks, we merge the method- and statement-level dependence to form the expanded inter-procedural statement-level dependence. During the merging process, we follow the classic approaches to additionally extracting parameter-in, parameter-out, and summary dependence [28]. Specifically, for every method invocation, we record key information including the call site, the callee, the actual-in and actual-out statements at the call site, the formal-in and formal-out statements in the callee procedure, and so on. After collecting these elements, we just add statement invocation, parameter-in, parameter-out, and summary dependence according to their definitions. For instance, whenever a method is invoked, the statement invocation dependence between the invoking statement and the entry node of the invoked method is added to the dependence graph. Due to space constraints, we do not provide detailed definitions of parameter-in, parameter-out, and summary dependence. Readers are encouraged to refer to the paper [28] for a deeper understanding of these concepts in the context of program slicing.

Thus we re-define the expanded statement-level dependence as follows:

Definition 5. *The **expanded statement-level dependence** denoted by R_{S+} is an expanded set based on the **statement-level dependence** R_S, which **additionally** contains*

- *the **statement invocation relation**. For $A, B \in S_R$, statement A is the call site statement and statement B is the entry statement of the called method.*
- *the **parameter-in relation**. For $A, B \in S_R$, statement A is the actual-in statement at call-site and statement B is the corresponding formal-in statement at callee.*
- *the **parameter-out relation**. For $A, B \in S_R$, statement A is the formal-out statement at callee, and statement B is the corresponding actual-out statement at call-site.*
- *the **summary relation**. For $A, B \in S_R$, statement A is the actual-in statement, and statement B is the actual-out statement at the same call site.*
- *the **statement field-data relation**. For $A, B \in S_R$, statement A defines field variables used in statement B.*

*4) **MDG Construction**:* As constructing CDG is complex and time-consuming, we design a series of strategies during the multi-layer dependence analysis process. Based on this analysis, we can construct a more concise dependence graph structure named the multi-layer dependence graph (MDG). We define the MDG as follows:

Definition 6. *Given a source-sink API list L, a **multi-layer dependence graph (MDG)** for app $\mathcal{P}$ is denoted by $G_M = \langle B_M, V_M, R_{S+}, E_M \rangle$, where*

- *B_M denotes the narrowed analysis scope in the target program, which is a tripe $\langle S_R, M_R, C_R \rangle$. Set S_R, M_R, and C_R denotes the set of L-related statements, methods, and classes in $\mathcal{P}$ respectively.*
- *V_M is a set of nodes. Each node $v \in V_M$ is tripe $\langle s, m, c \rangle$, which represents a statement $s \in S_R$ in the method $m \in M_R$ and the class $c \in C_R$.*
- *R_{S+} is the **expanded statement-level dependence**.*
- *E_M is a set of directed edges, which satisfies $E_M \subseteq V_M \times V_M$.*

According to the MDG definition, we hierarchically extract class-, method-, and statement-level dependence and merge

them into inter-procedural statement-level dependence. The MDG is constructed by adding these dependence as edges between statement nodes. By identifying class-level dependence and hotspot classes, we effectively reduce the size of the node and edge set, generating a smaller dependence graph than the CDG. Finally, we merge method- and statement-level dependence to complete the MDG construction.

C. Debloating

After constructing the MDG, we use it as input for the debloating stage. This stage consists of two phases: the chopping phase, which applies a program chopping algorithm to slice the MDG and corresponding code in the app, and the supplement phase, which restores necessary code elements to ensure the debloated app remains analyzable by static analysis tools.

Program Chopping. Given the source-sink API, our approach sets the source-sink invocation list as the chopping criteria. Following the definition of chopping on a dependence graph [16], we apply a vertex-reachability algorithm on the MDG to obtain a chop. Specifically, we alternately perform backward slicing from the to-criteria and forward slicing from the from-criteria to construct a chopped MDG. Since the dependence graph represents the target program, removing a node in the graph naturally corresponds to eliminating a statement in the actual program. Thus, we utilize the chopped MDG as a reference to remove the corresponding code elements and debloat the target app.

Entry Points Supplement. Although the MDG slice preserves all necessary dependence from source to sink invocations, this does not guarantee that the debloated app can be successfully analyzed by static analysis tools. In most cases, the forward-slicing step in the chopping phase omits entry points, causing the app to fail analysis checks. To address the multi-entry characteristics of the Android framework, we extract all component classes from the manifest file and collect their lifecycle and callback methods, along with other callbacks in non-component classes. By leveraging reaching definition analysis, we identify statements with data dependencies on entry methods and restore them in the chopped MDG, ensuring compatibility with static analyzers.

IV. Evaluation

We have implemented the debloating approach in our tool ***FlowSlicer***, which debloats Android apps with the MDG construction module and the program chopping module. FlowSlicer is built on top of the bytecode transforming framework *Soot* [32], which supports re-writing Dalvik bytecode with the tool *Dexpler* [7]. In the MDG construction module, it adopts *FlowDroid* [6] to build the dummyMain class for method-level dependence extraction. FlowSlicer also reuses the data-flow summaries for Android libraries provided by tool *StubDroid* [5]. FlowSlicer supports a user-customized source-sink API list and maximum execution time, which are configured by default as the *SourcesAndSinks.txt* file in FlowDroid and 60 minutes respectively.

We conducted a comprehensive review of existing Android app debloating tools and identified several representative approaches, including *XDebloat* [30], *Autodebloater* [19], and *Minimon* [20]. These tools typically rely on user-specified entry points or heuristics to remove unused code. While effective at reducing app size or simplifying app functionality, they are not designed to preserve source-to-sink data-flow paths and generally lack awareness of the data-flow semantics required for precise taint analysis. We believe that including these general-purpose tools in our evaluation would produce misleading results: changes in the number of detected leaks may reflect arbitrary removal of taint-relevant code, rather than actual improvements in analysis precision or performance. Therefore, we excluded such tools to maintain a fair and meaningful comparison centered on taint-aware debloating.

A. Evaluation Setup

Target Static Analyzer. Our evaluation is designed to figure out whether FlowSlicer can promote static taint analysis through the debloating approach. As FlowDroid [6] is one of the state-of-the-art Android static taint analyzers [22], [26], [38], we chose it as the target static analyzer to evaluate the performance of FlowSlicer. FlowSlicer is tool-agnostic and operates at the APK level, removing code unrelated to source-sink flows. While we use FlowDroid as the target analyzer, other tools like Amandroid [33] and DroidSafe [13] can analyze the debloated apps as well and benefit from the reduced analysis scope.

Research Questions. Overall, our evaluation is driven by the following research questions:

- RQ1: How effective is FlowSlicer in retaining static analysis results?
- RQ2: How efficient is FlowSlicer in debloating real-world applications?
- RQ3: How much code can be removed by FlowSlicer?

Timeout Threshold. To conduct our experiments, we employed FlowDroid to perform taint analysis on the real-world Android apps, using a timeout threshold of 120 minutes. Selecting an appropriate timeout value is non-trivial, as analysis times can vary significantly across different apps. According to our survey of prior work, existing studies typically configure FlowDroid with time budgets ranging from 30 to 120 minutes [22], [26], [38]. To empirically determine a practical timeout setting, we conducted exploratory experiments on real-world apps using four different timeout thresholds: 60, 90, 120, and 150 minutes. Among these, the 120-minute configuration offered the best trade-off between analysis completeness and resource efficiency. Based on this observation, we adopted the 120-minute timeout for our large-scale evaluation.

Benchmarks. For both RQ1 and RQ3, we take DroidBench2.0 [2] as our benchmark. DroidBench 2.0 is an open-source test suite designed to evaluate the effectiveness of Android taint-analysis tools. It consists of 119 hand-crafted apps, each tailored to assess various aspects of static analysis, including fundamental taint-analysis challenges, Android-

specific complexities, etc. For RQ2 and RQ3, we also select the real-world Android app benchmark constructed by Luo et al. [22]. After utilizing FlowDroid to perform taint analysis on the real-world apps with a 120-minute timeout, we found out that 174 out of 1,022 apps finally triggered the timeout.

Due to the substantial time required for a single analysis run, we sorted all the 1,022 apps alphabetically and divided them based on whether FlowDroid could complete analysis on the app within 120 minutes. From each subset, we alphabetically sampled 40 apps to form the COVABench, ensuring diversity and avoiding size bias. Specifically, the COVABench consists of 80 apps in two subsets:

- COVABench$_1$ contains 40 apps (4.1MB on average) selected from the 848 analyzed apps under FlowDroid.
- COVABench$_2$ contains 40 apps (5.8MB on average) selected from the 174 apps triggering a 120-minute timeout under FlowDroid.

The DroidBench and the COVABench together provide a comprehensive basis to validate FlowSlicer's effectiveness across different analysis scenarios.

Experimental Environment. All experiments are conducted on a server equipped with dual Intel(R) Xeon(R) Gold 6133 CPUs, each running at 2.50 GHz with 20 cores and 40 threads per CPU, providing a total of 40 physical cores and 80 threads. It also features 512 GB of DDR4 memory, using eight Samsung 64 GB modules.

B. RQ1: How effective is FlowSlicer in retaining static analysis results?

1) Effectiveness on DroidBench: For RQ1, the evaluation is firstly conducted on the DroidBench dataset, which equips the apps with corresponding leak oracles to verify whether the taint analyzer correctly identifies the tainted data-flow leaks. After checking and confirming the leak oracles of all apps, we found that the leak oracles recorded in DroidBench 2.0 are incomplete and just 117 out of 119 apps are provided with leak oracles. Thus we inspected the source code of the remaining apps and added oracles for the left leaks manually. For evaluation, we measure the performance of FlowDroid by recording the detected leaks in the scenarios of using FlowDroid only and using FlowDroid cooperated with FlowSlicer, i.e., we take two types of the input app for FlowDroid, one is the original Android app $\mathcal{P}$, and the other is the debloated app $\mathcal{P}'$. The results of detected leaks by FlowDroid in both cases are listed in Table I.

TABLE I
DETECTED LEAKS BEFORE AND AFTER DEBLOATING

Benchmark	#App	Input	#Leak	#TP	#FP	Prec.
DroidBench	119	$\mathcal{P}$	94	73	21	0.777
		$\mathcal{P}'$	86	73	13	0.849

Table I represents the number of detected leaks as well as the true positives (TP) and false positives (FP) annotated according to the leak oracles, exhibiting FlowSlicer's capabilities of preserving tainted data flows and improving taint analysis precision. Before debloating, FlowDroid detected a total of 94 leaks, achieving 73 true positives, but misidentifying 21 false positives. After applying FlowSlicer, FlowDroid identified 86 leaks, successfully retaining all 73 TPs while reducing FPs from 21 to 13, reaching a 38.09% decrease. The results also show a clear improvement in precision ($TP/(TP+FP)$), rising from 0.777 to 0.849 when FlowSlicer is applied. The enhancement of FlowSlicer's debloating effects on FlowDroid is two-fold: FlowSlicer is capable of retaining data flows between identified sources and sinks due to its precise modeling of dependence in MDG. What is more, the precise source-sink identification step in FlowSlicer reduces the number of possible false positive leaks detected by FlowDroid.

After manually checking and reviewing the false positive leaks eliminated by FlowSlicer, we found that 7 out of the 8 reduced false positive leaks are related to correct source and sink APIs identification. It is worth mentioning that for apps equipped with source-sink invocation statements but without permission declaration, no leaks will occur in actual execution scenarios. Flowdroid did not pay attention to the required permission check when pre-matching source and sink APIs, so the APIs lacking necessary Android permissions are also labelled as sources or sinks. Fortunately, our tool performs a separate effective source-sink identification before debloating, thus successfully removing the invalid sources and sinks.

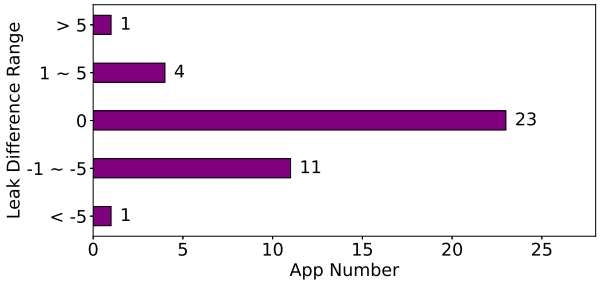

Fig. 2. App Distribution by Leak Difference Range on COVABench$_1$

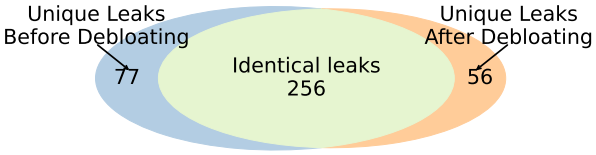

Fig. 3. Venn Diagram of Unique and Identical Leaks Before and After Debloating on COVABench$_1$

2) Effectiveness on COVABench$_1$: For real-world apps, we pick COVABench$_1$ to evaluate whether the identified leaks remain consistent after debloating, as these apps do not trigger timeout. The results of detected leaks on real-world apps before and after debloating are listed in Fig. 2 and Fig. 3.

As shown in Fig. 2, the bar chart represents the number of apps whose detected leak difference after debloating lies in the ranges of less than -5, from -1 to -5, 0, from 1 to 5, and more than 5. It is evident that for most apps (23 out of 40), the

number of leaks remain unchanged after debloating and the number of apps whose leaks increased after debloating is less than those decreased respectively (5 vs. 12). Specifically, the venn plot in Fig. 3 summarizes the number of identical and unique leaks before and after debloating on COVABench$_1$. We conducted a comprehensive analysis and deduplication of all leaks detected across the apps in Fig. 3, yielding a total of 389 leaks. Among them, 65.81% (256 out of 389) were identical leaks before and after debloating, while 56 leaks were newly identified. Additionally, 77 leaks disappeared after the debloating process. As the default reports do not include leak paths, we collected this information by rerunning under FlowDroid's "debug mode" and manually inspected the leak paths that can be re-collected under this mode (23 leaks in total). We found that 19 missed leaks were false positives and 4 were wrongly removed true positives, which means FlowSlicer may bring unexpected code removal in a few cases.

Specifically, the key limitations causing the unexpected precison loss stem from the precision of MDG construction and the design of the slicing strategy. Since the MDG is built statically, it cannot capture all runtime behaviors. This introduces a fundamental tradeoff: a more aggressive slicing strategy yields greater code reduction but may risk removing taint-relevant paths, while a conservative strategy better preserves precision but offers less debloating benefit. We also confirmed that the 56 newly identified leaks are true positives. Therefore, as FlowSlicer can help to report more new true leaks, it is still an effective tradeoff in static analysis.

After demonstrating the effectiveness of FlowSlicer, we would also like to further illustrate the safety of the tool. In the context of FlowSlicer, safety refers to the preservation of taint-relevant code for static analysis. We define safety along two dimensions: (1) analyzability—whether the debloated app remains analyzable; and (2) correctness—whether the tool preserves true positive leaks by analyzing the debloated app. In our experiments, all debloated apps remained analyzable, and FlowSlicer preserved the vast majority of real leaks.

Answer to RQ1: FlowSlicer is capable of not only retaining the leaks detected by the static analyzer FlowDroid but also improving the analysis precision in certain cases. On DroidBench, it successfully identifies all of original TP leaks and reduces 38.09% FP leaks compared to the original results. On real-world apps, it primarily achieves high consistency in identifying leaks before and after debloating.

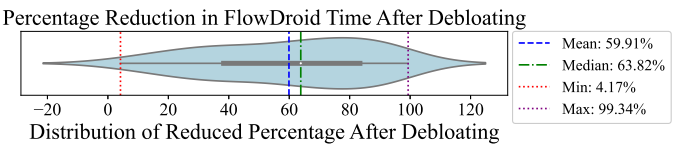

Fig. 4. FlowDroid Time Reduction Before and After Debloating for Apps in COVABench$_1$.

C. RQ2: How efficient is FlowSlicer in debloating real-world applications?

For RQ2, the evaluation is adopted on both COVABench$_1$ and COVABench$_2$. Generally speaking, the leak detection involves two main procedures: executing FlowSlicer to debloat the target app; and running FlowDroid on the generated app. To verify the efficiency, we record the time consumed by the two procedures and observe whether these apps successfully execute these two processes within the preset 120-minute timeout. For apps in COVABench$_1$, the execution time of FlowDroid before debloating is available, thus enabling us to compare the difference of FlowDroid time before and after debloating. The average execution time of FlowSlicer on COVABench$_1$ is 5.61 minutes. The results of FlowDroid time difference on COVABench$_1$ are listed in Fig. 4.

According to the results in Fig. 4, the execution of FlowSlicer significantly reduces the analysis time required by FlowDroid for apps in COVABench$_1$. The percentage reduction in time consumption ranges from a minimum of 4.17% to a maximum of 99.34%. Despite the wide variation in time reduction percentage across individual apps, both the median (63.82%) and mean (59.91%) time reduction rates indicate a substantial improvement in efficiency. The wide range of analysis time reduction can be attributed to differences in app complexity, code redundancy, and taint-relevant logic distribution. Apps with a large proportion of irrelevant third-party libraries or loosely connected components tend to benefit more from our debloating process. In contrast, lightweight or densely taint-connected apps exhibit more modest gains due to limited optimization potential. Fig. 4 also visually demonstrates that the majority of apps experience considerable time savings. Moreover, when aggregating the total FlowDroid runtime across all evaluated apps, we observe a 67.29% reduction after debloating, further confirming FlowSlicer's effectiveness in accelerating static taint analysis at scale.

Apart from COVABench$_1$, we also evaluate the efficiency of FlowSlicer on COVABench$_2$. According to the results on COVABench$_2$, we find out that 6 of 40 real-world apps successfully finished the leak detection task in 120 minutes. Among these, FlowDroid identified 156 new leaks when analyzing the debloated apps with FlowSlicer, compared to directly analyzing the original apps. Table II gives the number of statements, execution time of FlowSlicer, execution time of FlowDroid, and the number of newly detected leaks for the six apps managing to pass the analysis. To provide an overview of the target app size, we record their Jimple statements and list them in Table II. The average statement number is 130.45K. The execution time consumed by FlowDroid (T_{FD}) and FlowSlicer (T_{FS}) varies from less than 1 minute to about 40 minutes, showing great instability in the results which depend on the basic conditions of apps under analysis. For example, although the FlowDroid and FlowSlicer times for the last two apps are significantly shorter than those of the first four apps, our examination on their debloating results revealed no issues in the debloating process. After debloating,

these apps successfully triggered leaks that FlowDroid initially missed, further demonstrating that FlowSlicer's code reduction effectively lowered FlowDroid's analysis complexity. Finally, there are a total of 156 leaks newly detected by FlowDroid with the help of FlowSlicer, indicating that the debloating process not only enhances the efficiency but also improves the accuracy of FlowDroid's leak detection, as previously obscured or overlooked leaks become detectable.

TABLE II
DETAILS OF APPS THAT CAN DETECT NEW LEAKS AFTER DEBLOATING

App Name	#Stmt (K)	T_{FS} (min)	T_{FD} (min)	#Leak
MassagersCou	261.79	42.20	12.27	60
Eccentric lady	112.68	11.27	15.17	24
Downloader	119.45	8.17	26.08	19
CatClinic	262.09	38.41	11.05	49
Santa Doroteia Porto Alegre	12.91	0.25	0.03	2
TainData Tierra	13.79	0.24	0.02	2

Answer to RQ2: FlowSlicer is capable of debloating real-world applications efficiently on both COVABench$_1$ and COVABench$_2$. The debloating process substantially decreases FlowDroid's runtime while simultaneously revealing previously undetected leaks, underscoring its dual benefit of enhancing both efficiency and precision in static analysis of Android apps.

D. RQ3: How much code can be removed by FlowSlicer?

For RQ3, we evaluate the code removal performance of FlowSlicer on DroidBench, COVABench$_1$, and COVABench$_2$. The code removal amount of FlowSlicer is relevant to the size of input source-sink API list, and we adopt the default *SourcesAndSinks.txt* file provided by FlowDroid to conduct our experiments. After running FlowSlicer to debloat apps in our benchmarks, we calculate the ratio of the removed statements in the generated app to original statements to measure the code removal capability of FlowSlicer. Results are displayed in Fig. 5, where the recorded values are the relative size of the removed statements in percentage.

According to Fig. 5, for DroidBench, the distribution of removed statement percentages is centered around a higher range compared to COVABench$_1$ and COVABench$_2$. For DroidBench, the removed statement percentages of half apps fall within 55.48%-66.33%. The median value is around 60.77% and the highest ratio is 82%. Although DroidBench shows a higher average code removal ratio, it exhibits considerable variability, with code removal ratios ranging from as low as under 40% to as high as around 80%. Compared to it, COVABench$_1$ and COVABench$_2$ possess a relatively lower range of removed statements due to their more complicated app scales. Between two COVABenches, COVABench$_2$ removed only 10.42% statements in the mean and the maximum reduction ratio is about 38%, while COVABench$_1$ showed a higher mean statement removal percentage of 25.27% and an overall higher half-app-range from 19.78% to 28.15%, indicating a generally higher code removal amount in COVABench$_1$. The key reason lies in the fact that apps in COVABench$_2$ are selected due to their inability to complete FlowDroid analysis within a 120-minute timeout, revealing their averagely larger size and more complex internal dependence compared to apps in COVABench$_1$. When debloating COVABench$_2$ apps, FlowSlicer often encounters cases where it cannot complete debloating within the maximum execution time. As a result, it adopts a relatively conservative code removal range to ensure the generation of debloated apps, significantly reducing the removed statements ratio. Overall, FlowSlicer achieved an average code removal ratio of 60.77% on DroidBench and 17.85% on COVABench$_1$ and COVABench$_2$, demonstrating FlowSlicer's exceptional performance in code reduction.

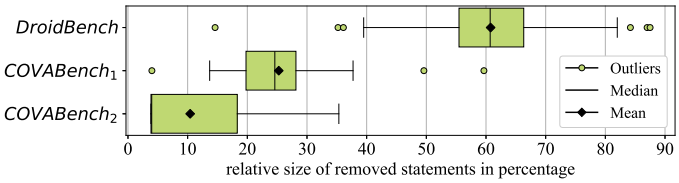

Fig. 5. The Removed Statement Ratio in DroidBench, COVABench$_1$, and COVABench$_2$.

Answer to RQ3: FlowSlicer is capable of removing unnecessary statements significantly on both the DroidBench (60.77%) and real-world (17.85%) applications.

V. THREATS TO VALIDITY

In this section, we illustrate the main threats to the validity of our approach.

We have applied several approximations to the implementation of our debloating approach. For example, Callback classes are identified on the basis of FlowDroid's default list of callback classes in our tool. Callback methods are identified in general by a name prefix ("on"). Our tool relies on the static analyzer StubDroid [5] to automatically construct data-flow summaries from Android library methods, thus accomplishing the complete define-use chain analysis when generating the program dependence graph for every method. When there is no available StubDroid summary (although generatable) for a given library method, it is assumed that neither the base nor a parameter of the method is reassigned by the invoked method, thus resulting in the inaccuracy of the built dependence graph and the incompleteness of the chopped app.

The correctness of our evaluation results to the DroidBench set depends on the correctness of their publicly available leak oracles. We have also manually checked the two apps without leak oracles and re-added the necessary leak oracles, which, however, may cause errors in annotations if we do not understand the author's intention in designing the app leak. As for the static taint analyzer for Android, we choose FlowDroid but not other tools to evaluate our approach. Due to the limited length of the paper and experimental resources, we can

only continue to conduct experiments on other static analysis tools in our future work to launch a more comprehensive and diversified evaluation of the effectiveness of our method.

Our tool does not support apps that have been obfuscated. The main problem comes from the source-sink identification step. If an app is obfuscated, the source-sink identification may fail to locate source and sink invocations correctly, especially for the permission-oriented API items provided by FlowDroid, because this location approach relies on the method signatures for locating source and sink APIs in an app.

VI. Related Work

A. Program Debloating

Previous work on program debloating sheds light on how to prune unused functions in a program. Soto et al. [29] utilized several Java bytecode coverage tools to precisely capture code dependence when running with a specific workload and thus implemented coverage-based debloating on java projects. Tang et al. [30] supported different feature location methods and debloated apps at various granularity. Liu et al. [19] proposed an ATG-based debloating approach to remove activities selected by users in Android apps. Bruce et al. [8] leveraged augmented static reachability analysis to debloat Java bytecode, which was capable of dealing with different dynamic features (e.g. Reflection, Invokedynamic, JNI, etc.) in Java. Based on this work, Macias et al. [23] provided rich visualizations of the bloated code elements within the target software project. Different from these studies, our work not only focuses on removing unused functional features from the perspective of users but also performs analyzer-oriented debloating steps on Android apps.

B. Program Slicing

Program slicing is a classical program analysis technique that extracts the parts of a program relevant to a specific computation, facilitating tasks such as debugging, refactoring, vulnerability detection, and large-scale information flow analysis. Over the years, it has evolved into a versatile tool adapted across diverse domains, including program repair, visualization, graph processing, etc.

From a foundational perspective, Zhang et al. [41] proposed a novel formalism based on the modular monadic semantics of programming languages. By modeling slicing as a slice monad transformer, their approach enabled semantic-level integration of slicing into language descriptions, providing a basis for more principled slicing frameworks that abstract away from intermediate representations. To further improve the efficiency and scalability of slicing in practice, Zhang et al. [40] introduced SymPas, a lightweight symbolic slicing technique that avoids repeated reanalysis of procedures by generating parameterized symbolic slices. Compared to traditional SDG-IFDS-based slicing, SymPas demonstrates significant reductions in both time and space costs in large-scale applications.

In the area of program repair and vulnerability detection, program slicing is often used to localize contextually relevant or irrelevant statements. Zhang et al. [42] utilized slicing and dependence analysis to extract bug-centered code contexts, which were then used to guide learning-based repair. Similarly, Cai et al. [9] leveraged slicing to eliminate irrelevant segments in smart contract functions, improving the precision of vulnerability detection.

Beyond code-level analysis, slicing has also shown great potential in structuring large-scale data. Li et al. [18] applied a hierarchical slicing strategy to visualize academic graphs containing millions of nodes. Their system divided the graph into manageable slices and applied overlap-removal algorithms, improving scalability without compromising layout clarity. Wu et al. [36] further extended the slicing principle to image segmentation, modeling sliding windows and internal pixels as nodes in a graph to extract semantic regions using graph convolutional networks. In temporal networks, Zhang et al. [39] proposed a segmentation-based time slicing technique that grouped temporal edges with identical optimal-path characteristics. This grouping significantly promoted the efficiency of computing centrality metrics such as betweenness and closeness in dynamic graphs.

Taken together, these works demonstrate the breadth and adaptability of program slicing—from the formal foundations and lightweight symbolic methods to performance-critical systems analysis and visual processing. While most prior approaches focus on improving the modularity, efficiency, or scalability of slicing, our work departs from a different perspective: we utilize slicing as a cooperative mechanism that interweaves with taint flow analysis. By coupling slicing with a precise MDG and the backward-forward-combined traversal strategy, FlowSlicer achieves both improved leak detection accuracy and reduced analysis overhead.

VII. Conclusions

We propose a novel debloating approach based on multi-layer dependence graph and develop FlowSlicer to automate this process. FlowSlicer performs multi-layer dependence graph construction and program chopping to generate the chopped MDG and debloat Android apps according to the chop. FlowSlicer is competent for cooperative analysis with static taint analyzers. We evaluate FlowSlicer with the static taint analyzer FlowDroid on hand-crafted and real-world apps, and the experimental results show that FlowSlicer is capable of not only retaining true positive leaks but also enhancing the precision of static analyzers through eliminating false positives.

VIII. Data Availability

FlowSlicer's replication package, including the benchmark and the source code, is publicly available at https://github.com/SQUARE-RG/FlowSlicer.

IX. Acknowledgment

The authors would like to thank the ICSME 2025 reviewers for their helpful suggestions. This work is supported partially by the National Natural Science Foundation of China (NSFC) under grant number 62132020 and 62272465, and in part by the Major Project of ISCAS (ISCAS-ZD-202302).

References

[1] Wechat. https://baijiahao.baidu.com/s?id=1801185893703085221&wfr=spider&for=pc, 2023. 2023-10-02.

[2] Droidbench. https://github.com/secure-software-engineering/DroidBench, 2024.

[3] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools*. pearson Education, 2007.

[4] APKPure. Wechat versions. https://apkpure.com/wechat/com.tencent.mm/versions, 2024. Accessed: 2024-8-30.

[5] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, pages 725–735, 2016.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, 2014.

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38, 2012.

[8] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 135–146, 2020.

[9] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software*, 195:111550, 2023.

[10] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*, pages 77–101. Springer, 1995.

[11] Android Developers. Reduce your app size. https://developer.android.com/topic/performance/reduce-apk-size, 2024. Accessed: 2024-9-30.

[12] Google Support. Google play developer policy center. https://support.google.com/googleplay/android-developer/answer/9859372?hl=en, 2024. Accessed: 2024-9-30.

[13] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[14] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[15] Daniel Jackson and Eugene J Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 1994.

[16] Daniel Jackson and Eugene Joseph Rollins. *Chopping: A generalization of slicing*. Citeseer, 1994.

[17] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, pages 189–199. IEEE, 2018.

[18] Qi Li, Xingli Wang, Luoyi Fu, Xinde Cao, Xinbing Wang, Jing Zhang, and Chenghu Zhou. Vsan: A new visualization method for super-large-scale academic networks. *Frontiers of Computer Science*, 18(1):181701, 2024.

[19] Jiakun Liu, Xing Hu, Ferdian Thung, Shahar Maoz, Eran Toch, Debin Gao, and David Lo. Autodebloater: Automated android app debloating. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2090–2093. IEEE, 2023.

[20] Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. Minimon: Minimizing android applications with intelligent monitoring-based debloating. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[21] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1355–1367. IEEE, 2023.

[22] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 102–114. IEEE, 2019.

[23] Konner Macias, Mihir Mathur, Bobby R Bruce, Tianyi Zhang, and Miryung Kim. Webjshrink: a web service for debloating java bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1665–1669, 2020.

[24] Joanna McGrenere. "bloat" the objective and subject dimensions. In *CHI'00 extended abstracts on Human factors in computing systems*, pages 337–338, 2000.

[25] Felix Pauck and Heike Wehrheim. Jicer: Simplifying cooperative android app analysis tasks. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 187–197. IEEE, 2021.

[26] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.

[27] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[28] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, 1994.

[29] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. Coverage-based debloating for java bytecode. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–34, 2023.

[30] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. Xdebloat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering*, 48(11):4501–4520, 2021.

[31] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[32] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[33] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[34] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment*, 7(12):1191–1202, 2014.

[35] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[36] Zizhang Wu, Yuanzhu Gan, Tianhao Xu, and Fan Wang. Graph-segmenter: graph transformer with boundary-aware attention for semantic segmentation. *Frontiers of Computer Science*, 18(5):185327, 2024.

[37] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. ICCBot: fragment-aware and context-sensitive ICC resolution for android applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 105–109, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. Analyzing android taint analysis tools: Flowdroid, amandroid, and droidsafe. *IEEE Transactions on Software Engineering*, 48(10):4014–4040, 2021.

[39] Tianming Zhang, Jie Zhao, Cibo Yu, Lu Chen, Yunjun Gao, Bin Cao, Jing Fan, and Ge Yu. Labeling-based centrality approaches for identifying critical edges on temporal graphs. *Frontiers of Computer Science*, 19(2):192601, 2025.

[40] Ying-Zhou Zhang. Sympas: symbolic program slicing. *Journal of Computer Science and Technology*, 36(2):397–418, 2021.

[41] Yingzhou Zhang, Baowen Xu, and JE Labra Gayo. A formal method for program slicing. In *2005 Australian Software Engineering Conference*, pages 140–148. IEEE, 2005.

[42] Yuwei Zhang, Ge Li, Zhi Jin, and Ying Xing. Neural program repair with program dependence analysis and effective filter mechanism. *arXiv preprint arXiv:2305.09315*, 2023.