XDebloat: Towards Automated Feature-Oriented App Debloating

Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai,

Abstract—Existing programming practices for building Android apps mainly follow the "one-size-fits-all" strategy to include lots of functions and adapt to most types of devices. However, this strategy can result in software bloat and many serious issues, such as slow download speed, and large attack surfaces. Existing solutions cannot effectively debloat an app as they either lack flexibility or require human efforts. This work proposes a novel feature-oriented debloating approach and builds a prototype, named *XDebloat*, to automate this process in a flexible manner. First, We propose three feature location approaches to mine features in an app. XDebloat supports feature location approaches at a fine granularity. It also makes the feature location results editable. Second, XDebloat considers several Android-oriented issues (i.e., callbacks) to perform a more precise analysis. Third, XDebloat supports two major debloating strategies: pruning-based debloating and module-based debloating. We evaluate XDebloat with 200 open-source and 1,000 commercial apps. The results show that XDebloat can successfully remove components from apps or transform apps into on-demand modules within 10 minutes. For the *pruning-based* debloating strategy, on average, XDebloat can remove 32.1% code from an app. For the *module-based* debloating strategy, XDebloat can help developers build instant apps or app bundles automatically.

Index Terms—Android, Android App, Debloating, Instant App, App Bundle

1 Introduction

To fulfill the requirements from different users and mobile platforms, developers nowadays tend to build complex and large apps by including all features and supporting different versions of libraries and various application binary interfaces (ABIs) for different platforms in one app [1]. A *feature* in our context is defined as a functionality in an app that satisfies a certain requirement. For example, in a social media app, "chat with friends" is considered as a feature because it satisfies the communication requirement of a social media app. It is worth mentioning that the feature in our context is different from the one in software product line engineering (SPLE) [2] as features in a software product line are collected from a series of products instead of from a single product.

Such "one-size-fits-all" strategy usually results in software bloat [3], [4], which can introduce various problems to both users and developers, such as lower conversion rates, slower download speed, larger attack surfaces, and lower update rates [4].

Motivation and State-of-the-art. To address this issue, there are two types of solutions: *pruning-based debloating* and *module-based debloating*. The *pruning-based debloating* approach removes unwanted features in a program [5], [6], [7], [8], [9], [10]. For example, during software evolution, some functions or features become out-of-date and are no longer maintained. Developers may plan to prune them. Only one study aiming at Android apps adopts this strategy [6], which removes dead code during compilation and prunes redundancy ABIS, including multiple SDKs and embedded ABIs during installation. Our work significantly different from RedDroid [6] in two aspects: (1) **Code Redundant**. RedDroid only considers "dead code" as redundant. Besides removing dead code from apps, XDebloat is equipped with three feature location approaches to locate features in apps and also allows developers to define what to prune. (2) **ABIs Redundant**. In RedDroid, if the CPU architecture of a target device is ARMv5, the ABIs support ARMv7 (ABI: armeabi-v7a), x86 (x86), MIPS (mips), ARMv8 (arm64-v8a), MIPS64 (mips64), x86_64 (x86_64) are considered as redundant. The corresponding ABIs are removed by RedDroid. Besides removing redundant ABIs in apps, XDebloat also supports removing redundant resources from apps. There are many real-world needs for the pruning-based debloating approach. The *module-based debloating* solution suggests splitting an app into modules so that users can only download and install specific modules whenever needed. Following this idea, Google recently announced two frameworks: App Bundles [11], and instant-enabled App Bundles (a.k.a., Instant Apps, a specific type of app bundle [12]). Unfortunately, no automated tools are available for converting a legacy app into an app bundle or an instant app. The developers must re-engineer their apps manually, which is time-consuming and error-prone. Furthermore, Google announced that all new apps must be published with app bundle since August 2021 [11]. Thus, there is an increasing demand for a module-based debloating strategy that supports the app bundle format.

To address the above limitations, in this paper, we propose the *first* feature-oriented debloating framework for apps and develop a prototype named *XDebloat* to automate this process. In this paper, we focus on Android apps

Y. Tang is with ShanghaiTech University
H. Zhou, and X. Luo are with the Department of Computing, the Hong Kong Polytechnic University, Hong Kong SAR, P.R. China.
T. Chen is with University of Electronic Science and Technology of China
H. Wang is with Beijing University of Posts and Telecommunications
Z. Xu is with Chongqing University, China
Y. Cai is with the Institute of Software, Chinese Academy of Sciences
Y. Tang (tangyt1@shanghaitech.edu.cn) and X. Luo (csxluo@comp.polyu.edu.hk) are the corresponding authors.

as Android occupies over 80% market shares [13]. Given an Android app in bytecode format (i.e., Apk), XDebloat supports both pruning-based and module-based debloating.

Challenges. Debloating an app is challenging. **C1: type errors**: Type errors can be introduced to the debloated apps if they are not well resolved. For example, a debloated app can contain an invalid type usage if a type is removed, but its usage is preserved. Here, a "type" is the type concept in any object-oriented programming language. A type can be a primitive type (i.e., int, char, boolean) or a user-defined type. **C2: modelling unique dependencies in Android**: Unlike desktop applications, there are some indirect calls which are delegated by Android system. For example, Intents, in Android, can be leveraged to implement the communication between components (e.g., `Activity`). Therefore, to model the behavior of an app, such relations should be considered. Existing debloating approaches [8], [14], [5], [6] cannot address these challenges as most of them [8], [14], [5] are designed for C applications rather than Android apps. Though RedDroid [6] aims at Android apps, it fails to build a complete call graph for debloating. For example, since it does not handle inter-component communications [15] in an app, some methods are treated as dead code and debloated by mistake.

Solution. To address **C1**, when conducting the static analysis on apps, we pay attention to the unique mechanisms in Android, including data/control flows introduced by *implicit* Intent, and callbacks. We propose a type system to avoid introducing type errors during debloating. To address **C2**, we explore the Android-specific dependencies in an app in Sec. 3.2.

Contributions. This paper makes three major contributions:

- We conduct the *first* systematic investigation on feature-oriented app debloating;
- We develop a prototype named *XDebloat* for app debloating, which can not only prune unwanted (or undesired) features from an app but also can transform it into an instant app or an app bundle automatically; and
- We evaluate XDebloat on 200 open-source and 1000 commercial apps. The experimental results show that it can correctly debloat apps and generate instant apps/app bundles with low performance overhead. Moreover, it reduces the bloated code at the rate of 34.33% (with the `Activity`-based method), 29.31% (with the permission-based method), and 32.74% (with the modularity-based method), respectively. The related artifacts can be found at [16].

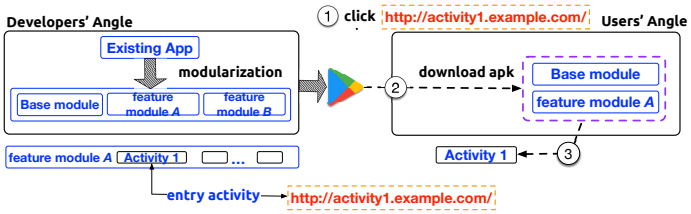

Fig. 1: The Workflow of Android Instant App.

2 Background and Motivating Example

Our feature-oriented debloating supports both *pruning-based* and *module-based* debloating. Specifically, in module-based debloating, we leverage two frameworks (instant app and app bundle) for module-based debloating. In Sec. 2.1 and 2.2, we present the workflows for these frameworks.

2.1 Instant App

An instant app allows users to access its functions without installing the entire app [12]. Each instant app consists of one *base module* and zero or more *feature modules*. A *base module* contains the fundamental functions in the app. The *feature modules* can access functions defined in the *base module*. Inside each module, there is at least one `Activity` that serves as the entrance to the module. This `Activity` is always associated with a URL, through which users can access the `Activity`. When a user clicks the URL, the module is downloaded and its `Activity` is shown to the user. If a user accesses a *feature module*, an apk that contains the *base module* and the *feature module* is downloaded. Otherwise, an apk that only contains the *base module* is downloaded [12], [17].

Example. Fig. 1 shows an example, where an app is reorganized into three modules, including one base module, and two feature modules `A` and `B`. `Activity 1` is the entrance of module `A`. When a user clicks http://activity1.example.com in an app (e.g., SMS app), the Android system downloads the feature module `A` and the base module from Google Play and then displays `Activity 1` to her/him.

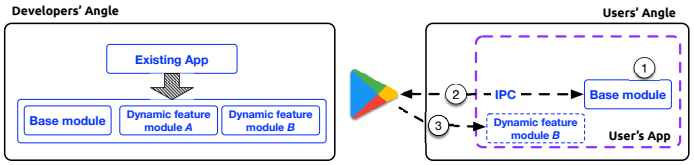

Fig. 2: Loading Dynamic Feature in App Bundle.

2.2 App Bundle

By organizing apps into modules, app bundle allows the modules in an app to be delivered to users on demand [11]. An app bundle *must* contain one base module. It is optional for an app bundle to have *dynamic feature modules* which are delivered to users on demand. As shown in Fig. 2, after the base and dynamic feature modules are compiled into an `.aab` archive, it can be uploaded to Google Play for distribution. Google Play uses the app bundle to generate apks for each device configuration, thus the code and resources that are needed for a specific device are downloaded at runtime. From developers' perspectives, they do not need to build multiple apks for different types of devices. From users' perspectives, they can get on-demand feature modules when needed.

Example. Users first download the base module from Google Play. When they access any function in the dynamic feature modules, the corresponding modules are then downloaded from Google Play. For example, the dynamic feature module *B* in Fig. 2 can be downloaded when users access it.

Different from *pruning-based* debloating, *module-based* debloating targets at splitting an app into modules and modules are delivery in an on-demand manner. The frameworks

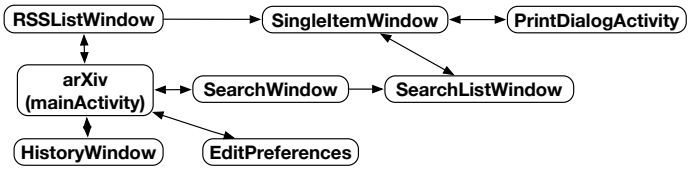

Fig. 3: Activities transition graph in the arXiv app.

we used (i.e., instant apps and app bundles) support such scheme, therefore they are qualified for *module-based* debloating tasks.

2.3 Motivating Example

We use the arXiv app (com.commonsware.android.arXiv) as an example to illustrate our feature-oriented debloating process in general. Fig. 3 shows all activities in the arXiv app and the transitions among them. After launching the app, users can view the feed list (*RSSListWindow*) and the history record (*HistoryWindow*), set up a preference for the app (*EditPreferene*), and search an article by constructing a query (*SearchWindow*). The search results are displayed in the *SearchListWindow*. If a specific item is selected, the details of this item are displayed on the *SingleItemWindow*. The app also allows users to print an article (*PrintDialogActivity*).

For the given app (i.e., arXiv), XDebloat performs the following steps:

Step 1 (Feature Location): For a given app, in this step, XDebloat locates its features and generates the corresponding configuration files for the next step. Identifying features in an app assists developers in exploring features that can be debloated. XDebloat supports feature location approaches at all granularities, including class, method, and statement. Developers can leverage XDebloat to annotate features in an app manually. We also support three approaches to automatically locate features in apps, including `Activity`-based approach, permission-based approach, and modularity-based approach. Specifically, the `Activity`-based feature location approach takes each `Activity` and all UI callbacks affiliated to this `Activity` as a feature. Therefore, with `Activity`-based approach, there are 8 features in the arXiv app as it contains 8 `Activities` (Sec. 3.3.1). The permission-based feature location approach determines features by inspecting the permissions required by the app. As the arXiv requires the `NETWORK` permission, the permission-based feature location can locate the code fragments that directly or indirectly require this permission. Those code fragments are considered as a feature. The permission-based feature location approach finds the statement "xr.parse(new InputSource(url.openStream()))" invokes the method "java.net.URL: java.io.InputStream openStream()", which requires the `NETWORK` permission (Sec. 3.3.2). The modularity-based feature location approach leverages the Louvain algorithm, which is a community-detection algorithm, to explore features. As shown in Fig. 8, the modularity-based feature location results in 10 features (Sec. 3.3.3).

Step 2 (Specification Construction and Checking): In Step 2, XDebloat generates a configuration in a human-readable format so that developers can easily modify it based on their needs. The configuration file (i.e., configuration.log) records a series of pairs $\langle$fId, config$\rangle$, each of which represents a mapping from a feature Id `fId` to its `config`. Note that each feature is assigned to a unique id named `fId` and the value of `config` can be 1 or 0. If the `config` of a feature is 0, XDebloat prunes it from the app. Note that the configuration is only used in pruning-based debloating. For module-based debloating, features in an app are transformed into modules instead of being pruned.

configuration.log

feature (Activity-based)	fId (feature Id)	configuration
RSSListWindow	1	1
SingleItemWindow	2	1
PrintDialogActivity	3	0
arXiv	4	1
SearchWindow	5	1
SearchListWindow	6	1
HistoryWindow	7	0
EditPreference	8	1

PrintDialogActivity should be pruned

HistoryWindow should be pruned

Fig. 4: Example of configuration.log

Fig. 4 gives an example of `configuation.log` file, which contains a list of features fetched by the `Activity`-based feature location approach. The `Activity`-based feature location returns 8 features. Here, we simply leverage the `Activity` name to represent each feature as shown in Fig.4. In the configuration.log, we leverage fId (i.e. feature Id) to represent each feature. The configuration.log stores the mapping between fIds and the corresponding configurations. In this paper, we term the configuration.log as a *configuration* for XDebloat. For example, the 0s in Fig. 4, feature `PrintDialogActivity` and `HistoryWindow` should be pruned, while others should be preserved.

However, for XDebloat, only has a *configuration* (i.e only knowing the features to prune) is not sufficient, XDebloat must explicitly know which programming element to prune. Therefore, we must build a corresponding *specification* for the configuration. Different from a *configuration*, a *specification* illustrates the mapping between a programming element (e.g., a method) to a *decision*. In this paper, we define three possible decisions, which are *remove*, *preserve*, and *unknown* (see Sec. 3.6). The *remove* indicates a programming element must be pruned. The *preserve* indicates a programming element must be preserved. The *unknown* is a temporary decision, which is used during the specification configuration, to indicate the final decision (i.e.*remove* and *preserve*) is not determined yet. The process of transferring a *configuration* to a *specification* is introduced in Sec. 3.4.

Step 3 (Debloating): Given the *specification*, XDebloat performs either pruning-based or module-based debloating. For the former, XDebloat removes programming elements according to the *specification* given and generates the debloated app. For the latter, XDebloat builds the instant apps or app bundles for developers based on the *specification*.

3 XDebloat

3.1 Overview

As shown in Fig. 5, XDebloat processes a given app through four steps: feature identification (Sec. 3.3), specification con-

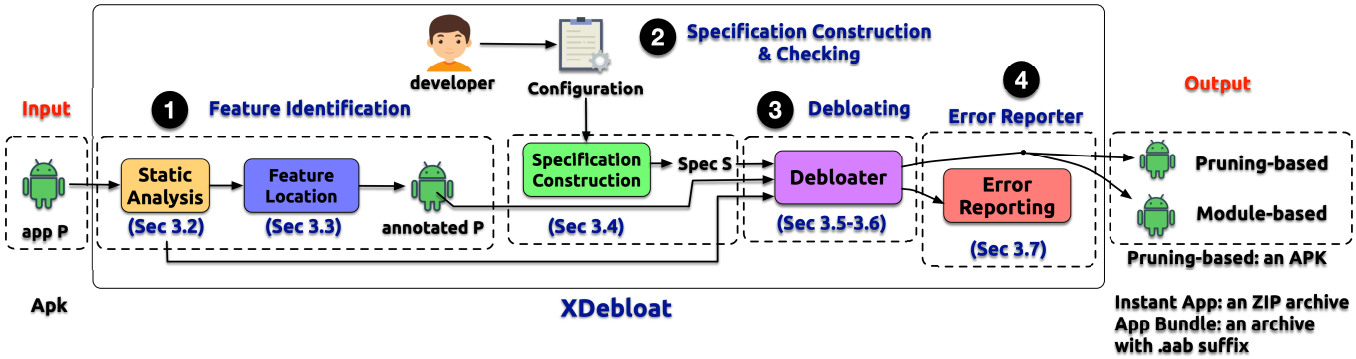

Fig. 5: Overview of feature-oriented debloating

struction and checking (Sec. 3.4), debloating (Sec. 3.5 and 3.6, and the error reporting system in XDebloat (Sec. 3.7).

The input app is in the binary format (i.e., apk). For pruning-based debloating, XDebloat produces a pruned app. For module-based debloating, XDebloat converts it to an instant app or an app bundle.

3.2 Static Analysis

3.2.1 Overview

We perform a static analysis on an app to collect the following information: all `Activities` in the `AndroidManifest.xml`, required permissions, all layout files, and program dependence graph (PDG) [18]. The PDG contains an app's data and control dependencies. `Activities` and permissions declared in `AndroidManifest.xml` are required for our `Activity`-based and permission-based feature location approaches. The PDG is used in feature location, specification, and debloating. The static analysis of an app starts from finding the entry points of the app. The entry points indicate where to start the static analysis.

3.2.2 Entry Points

Unlike Java applications, an Android app does not have an explicit entry point (e.g., `main` method). Therefore, we need to resolve the entry points in the app. The entry points of an app come from two sources [19]: (1) lifecycle methods in Android components (e.g., `Activity`); and (2) UI callbacks.

- *Android lifecycle methods.* In Android, *lifecycle methods* are the standard entry points to components, we first extract all components defined in the `AndroidManifest.xml` and then collect *lifecycle methods* declared in these components.
- *UI callbacks.* Developers can implement callback methods to capture UI events (e.g., button click). Each callback method is associated with a *registration* method [20], [21], which is explicitly declared by the app to communicate with the Android framework [21]. This is also known as *registration-callback* pairs [20]. The reason for mapping registrations to callbacks is that the invocations from registration methods to callbacks are indirect. By indirect, we mean the invocations are delegated by Android.

We leverage EdgeMiner [20] to explore all registration-callbacks in the Android framework. EdgeMiner takes the Android framework as the input to explore all *registration-callback* pairs supported by Android. However, for a specific app, it only contains certain *registration-callback* pairs. The reason that we adopt EdgeMiner is three-fold: (1) EdgeMiner is the state-of-art tool in finding UI callbacks; (2) EdgeMiner is widely adopted for the task of exploring the UI callbacks from apps [22], [23], [24]; (3) EdgeMiner offers precious results, which is proven with the DroidBench benchmark [20], [18]. Therefore, we can conclude that EdgeMiner is qualified for the task and it can provide a precise result.

3.2.3 Inter-component Communication (ICC)

Inside an app, data can be sent across different components through `Intents`. Since `Intents` can lead to additional control and data flows, we need to handle them as well. To infer the target of an `Intent`, we resolve it with the `IC3` [25], which transforms the ICC problem into a Multi-Valued Composite (MVC) constant propagation problem (i.e., finding all possible values of objects concerned at a certain program point) to compute the target of an `Intent`.

3.2.4 Reflections

It is also possible that some method invocations are introduced with Java Reflection. In FlowDroid, the method invocations introduced by reflection can be obtained by with the following setting: `setupApplication.getConfig().setEnableReflection(true)`. However, we find some reflection patterns that are not well supported by Soot, such as, `getFields()`, `loadClass()`. We migrate reflection analysis module in DroidRA[26], a state-of-art String analysis tool for reflection, into our framework. DroidRA transforms the resolution of reflective calls to a composite constant propagation problem. They leverage the COAL solver[15] to infer the target callee.

3.2.5 Asynchronous Task Dependencies

An asynchronous task (i.e., `android.os.AsyncTask`) is defined by a computation that runs on a background thread and whose result is published on the UI thread [21]. To implement an asynchronous task, developers can implement a subclass of an `AsyncTask` and overwrite the following methods: `onPreExecute`, `doInBackground`, `onProgressUpdate`, and `onPostExecute`. The task goes through 4 steps:

Step 1: `onPreExecute` is normally used to setup the task, which can be invoked by a UI thread with `AsyncTask.execute();`

Step 2: Once the `onPreExecute` finishes, the `doInBackground` is invoked on the background thread immediately by Android. Through the `doInBackground` method, developers can update the progress with `publishProgress`;
Step 3: After a call of `publishProgress`, the `onProgressUpdate` is invoked on the UI thread; and
Step 4: Once the background computation finishes, the `onPostExecute` method is invoked on the UI thread. The result of background computation is passed to this method with a parameter.

Based on the above steps, we can fetch the following call relations and append to our program dependency graph (PDG): 1) from `AsyncTask.execute()` to `onPreExecute` method; 2) from `onPreExecute` to `doInBackground`; 3) from `publishProgress` to `onProgressUpdate`; and 4) from `doInBackground` to `onPostExecute`.

3.2.6 Implementation

XDebloat is built atop FlowDroid [18], which provides a `dummyMain` method. The `dummyMain` method is a faked main method that allows traversing the call graph through `dummyMain` rather than all entry points. We link the entry point methods in an app to the default `dummyMain` method. Then, we use the IC3 framework to detect ICC in an app, and FlowDroid provides an interface (see class `Ic3ResultLoader`) for loading the result of IC3.

3.3 Feature Location

The first step of the feature-oriented debloating is to identify features in an app. By default, the feature location process is guided and conducted by developers as they have to determine the features in the app. It is worth mentioning that a feature can come from a functional requirement or a non-functional requirement [27].

To evaluate the debloating approach proposed, we propose three feature location solutions: `Activity`-based, permission-based, and modularity-based feature location approach. It is worth mentioning that we do not claim our feature location approaches as the oracle. As supported in existing studies [28], [29], [30], [31], different software needs different feature location approaches. There is no universe feature location approach that fits all projects. The reason we propose the feature location approaches is to evaluate the debloating framework. In practice, developers are also suggested to annotate features on their own instead of using our approaches with the toolkit in XDebloat. Also, developers can adjust the feature location results returned by XDebloat with the GUI toolkit offered by XDebloat.

3.3.1 `Activity`-based feature location approach

Definition. This method regards each `Activity` and all UI callbacks that are affiliated to the `Activity` as a feature. For example, Fig. 6 shows a sample of this feature.
Approach. XDebloat analyzes the layouts files to obtain the UI elements (e.g.,`findViewById`) and their callback methods (e.g., `OnClick`). Since an `Activity`'s layout can be changed at runtime, we conduct static analysis to extract all layouts that can be used by the `Activities` from code.

Since developers can put UI elements in a fragment and load a specific fragment at runtime, rather than displaying

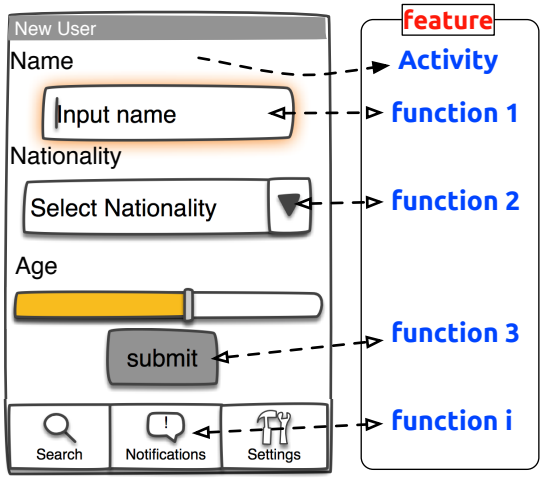

Fig. 6: Example of `Activity`-based feature location.

UI elements in an Activity, XDebloat determines all fragments that can be loaded into this Activity at runtime and groups them to the same feature.
Example. As aforementioned, in `Activity-based` feature location approach, each `Activity` and all UI callbacks that affiliated to the `Activity` as a feature. Recall the motivating example in Sec. 2.3, the arXiv app contains 8 `Activities`. Therefore, the `Activity`-based feature location approach identifies 8 features corresponding to 8 activities in the app. Each `Activity` (e.g., RSSList, SearchList, HistoryWindow) and all UI callbacks that affiliated to the `Activity` is considered as a feature.

3.3.2 Permission-based feature location

Definition. This method leverages the app permissions required by the app to locate features. Specifically, it groups the method calls requiring the same permission into a feature. If an app attempts to access some sensitive data (i.e., system setting) or use certain hardware (i.e., record audio), it must request the corresponding permissions. We focus on two types of permissions: privacy-oriented permissions and hardware-oriented permissions. We consider these permissions as features since they control whether apps can access certain sensitive data or use specific hardware. More precisely, we select 36 permissions, including all 26 dangerous-level permissions and 10 hardware-oriented permission. Hardware-oriented permissions are always associated with hardware on the Android device [32].

The 10 hardware-oriented permissions are INTERNET, BLUETOOTH, USE_FINGERPRINT, ACCESS_NETWORK_STATE, CHANGE_NETWORK_STATE, ACCESS_WIFI_STATE, CHANGE_WIFI_STATE, NFC, WRITE_VOICEMAIL, and READ_VOICEMAIL. The entire list of dangerous permissions is available on the project page.
Approach. We leverage PScout [33] to explore the method calls that require these permissions. Thus, we focus on whether a permission is dangerous or associates with specific hardware rather than whether the permission is a generic one or not. Therefore, even though the permissions, like "INTERNET", "ACCESS_NETWORK_STATE", can be generic, they are associate with the network adapter card.

The motivation for leveraging PScout is threefold: First, the PScout is the state-of-art tool in finding the mapping between Android APIs and permissions required. Second, based on the soundness evaluation from [33], PScout claims to be accurate. Besides, PScout provides a benchmark for the mapping between API calls and permissions. Last, PScout is widely adopted by other research works [34], [35], [36], [37] to explore the mapping between permissions and their corresponding APIs.

If an app requires a certain permission, XDebloat identifies all method calls that access this permission from PDG. If an app is with permissions beyond these 36 permissions, XDebloat does not consider the corresponding code as features. The reason is that for the other permissions, we do not find any indicator or evidence to consider them as features.
Example. Recall the arXiv example in Sec.2.3, since the arXiv app requires the `NETWORK` permission for accessing network, we regard it as a feature and locate code that require this permission in this app. As a result, XDebloat reports the statement "xr.parse(new InputSource(url.openStream()))" invokes the method "java.net.URL: java.io.InputStream openStream()", which requires the `NETWORK` permission.

3.3.3 Modularity-based feature location

Definition. This feature location approach is motivated by the fact that a program should try to make each module have high modularity [38], according to the principle of object-oriented programming [38]. As reported in [38], a program with high quality should make each module have high modularity. As suggested by [2], [29], [39], [40], in practice, a module in a program can be deemed to be a feature for users. Therefore, it is rational to propose a feature location approach based on the target of optimizing modularity. The modularity-based feature location approach separates an app into several modules concerning the overall modularity.
Approach. In practice, graph-based community detection aims at finding the communities based on the structure of the graph [41]. This method explores communities in the call graph and regards each community as a feature. More precisely, XDebloat transforms the method call graph of an app into an undirected graph G with weights. Note that we leverage "method call graph" instead of PDG for the following reasons: first, the method call graph is a sub-graph of PDG; second, in the module-based feature location approach, we focus on partitioning the app at the granularity of the method. As the features are extracted at the method granularity, we use the method call graph in this context instead of PDG. In G, each node represents a method in the app. Each edge indicates that there exists at least one call from one node to another. We leverage weights 0.5 and 1 to model unidirectional and bidirectional edges in G. Debloat runs the Louvain community detection algorithm [42] to partition G into communities. The motivation for choosing the Louvain community detection algorithm due to its advantages in two aspects: first, unlike other community detection algorithms, the Louvain community detection algorithm performs community detection in networks by maximizing a modularity function. For a program built with any object-oriented programming, reaching high modularity (i.e., high cohesion and low coupling) is the target. Therefore, the Louvain community detection algorithm matches the goal of optimizing the modularity of the whole program. That is, each module or each community explored by the Louvain algorithm holds high modularity. Each module or community explored by the Louvain algorithm is considered as a feature. Second, the Louvain algorithm can be applied to very large graphs (networks) within limited computing resources. For a large app, the number of methods can high. Therefore, an algorithm that suitable for solving large graphs can be qualified.

Example. For the arXiv app in Sec. 2.3, the Fig. 7 demonstrates how modularity-based feature location works on the arXiv app. First, the method call graph is extracted from the PDG. The undirected graph G is built with the call graph like the one shown in Fig. 7. Each node in G represents a method. Each edge represents an invocation relation. The weight of an edge can be 0.5 or 1. 0.5 stands for the method invocation that occurs in one direction (i.e., either m invokes n or n invokes m). 1 stands for the method invocation in both directions (m invokes n and n invokes m). In Fig. 7, we can show this with the weight of edges. A thick line shows the weight is 1, while the thin one shows the weight is 0.5. We also introduce "0" as a weight to indicate there is no edge between two nodes.

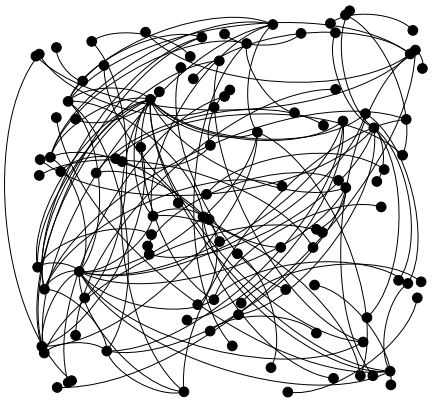

Fig. 7: Example arXiv app Call graph

We utilize the Louvain algorithm to this graph to divide it into different communities. In the modularity-based feature location approach, each community is considered as a feature. The result is shown in Fig. 8, in which different features are illustrated with different colors. The Louvain algorithm returns 10 features.

Comparison between feature location approaches. As our goal is proposing a feature-oriented debloating framework, proposing reasonable feature location approaches is a must. Therefore, we propose three feature location approaches, but we do not claim our approaches are the oracle. Different apps may require different feature location approaches. It can be hard to determine which feature location approach is the best or which one should be adopted. The feature location process should be under the supervision of developers. It is because what is a feature in an app is case by case. Only developers know the features of the app.

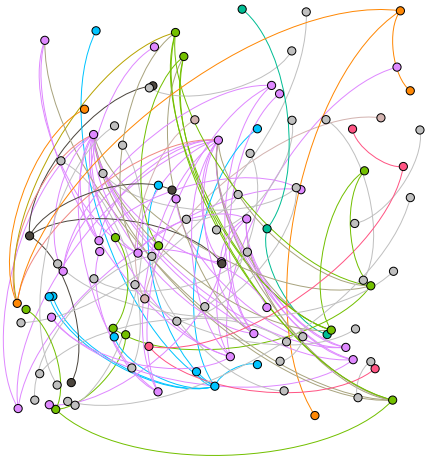

Fig. 8: Example of `Modularity`-based feature location.

TABLE 1: Language Model

Program	P	::= C*	
Class	C	::= M*	
Method	M	::= C m (s*)	
Statement	S	::= x := c	/* constant */
		| x := findViewById(i)	/* findViewById */
		| x := v	/* variable */
		| x := y.f	/* field access */
		| x := cast(y)	/* type cast */
		| x := x.m(t)	/* method invocation */
		| x := **new** C(t)	/* object creation */
		| x := return (x)	/* return */
Variable	x, v, y, t		/* variable */
ID	i		/* UI-related ids */

3.4 Specification Construction

3.4.1 Language Modeling

The debloating process may break the original typing relations between programming elements. To ensure the debloated app is well-typed and prevent unexpected types in the debloated app, we have to build a type system. The first step to build a type system is modeling the language. In this paper, we focus on apps in the Java language. Tab. 1 represents the abstract model language. A program consists of classes. A class contains a list of methods. A method is built with a list of statements.

We implicitly model the loop (e.g, `for`, `while`) statements as a single static assignment (i.e., $s = v$) as modeling the loop is irrelevant to our analysis. For the method invocation and object creation, we assume that there is only one argument for the method invocation expression (e.g., in `new C(t)`, there is only one argument t).

Besides, we set a special method invocation `findViewById` as it is leveraged to interpret a resource id to an instance of a UI object. For example, the expression `findViewById(R.id.button)` returns an instance of `Button` object from the resource id `R.id.button`.

3.4.2 Type System

The type system is a set of rules, which must be followed during debloating. For example, assuming that variable v is with type T in the original app, if variable v is preserved during debloating, our type system targets at describing what are the type requirements that v must follow. As for the capability, it holds the same capability as the Java's type system. From the implementation perspective, we obtained the type information from FlowDroid.

We define our type system as follows:

1) **T-NEW**:

$$\frac{\Gamma \vdash x : C_x \qquad C_x \in d\Gamma}{d\Gamma \vdash x = \text{new } C_x} \text{[TNEW]}$$

The **new** statement initializes an instance of type/class. Γ represents the empty context in the original app, and x is with type C_x. The expression $\Gamma \vdash x : C_x$ presents Γ proves x is with type C_x. $d\Gamma$ denotes the empty context in the debloated app. Therefore, if a type C_x is not removed during debloating (exits in the original and debloated app), the type of its instance (i.e., x) keeps consistent during debloating.

2) **T-ASSIGN**:

$$\frac{\Gamma \vdash x : C_x, y : C_y \qquad C_x, C_y \in d\Gamma \qquad C_y <: C_x}{d\Gamma \vdash x = y} \text{[TASSIGN]}$$

The **assignment** statement passes the result of right-side expression to left-side variable. In the debloated app, an **assignment** statement requires types of both right and left sides are compatible ($C_y <: C_x$).

3) **T-WRITE**

$$\frac{\Gamma \vdash x : C_x, y : C_y, f : C_f \qquad C_x, C_y, C_f \in d\Gamma \qquad C_x <: C_f}{d\Gamma \vdash y.f = x} \text{[TWRITE]}$$

The **write** expression $y.f = x$ assigns a value to a specific field. All types of involved variables y, f and x must be contained in the debloated app. The type system also ensures the type of x is a subtype of or same as $y.f$.

4) **T-READ**

$$\frac{\Gamma \vdash x : C_x, y : C_y, f : C_f \qquad C_x, C_y, C_f \in d\Gamma \qquad C_f <: C_x}{d\Gamma \vdash x = y.f} \text{[TREAD]}$$

The **read** expression $x = y.f$ obtains a value from a field, which requires the types of involved variables (y, f and x) exist in the debloated app. In addition, the type system also ensures the type of $y.f$ is a sub-type of or the same as x.

5) **T-INVK**

$$\frac{\Gamma \vdash x : C_x, y : C_y, z : C_z \qquad C_x, C_y, C_z \in d\Gamma \qquad type(m) = C_p \to C_{ret} \qquad C_{ret} <: C_x \qquad C_z <: C_p}{d\Gamma \vdash x = y.m(z)} \text{[TINVK]}$$

The **method invocation** $x = y.m(z)$ takes one argument z and the return is assigned to x. $type(m)$ returns the type of m, which includes type of parameter (C_p) and type of return value (C_{ret}). For simplicity, we only present method with a single parameter rather than two or more. More specific, the $C(p)$ is the type of method y's parameter p, which must be a super type of z. The return type C_{ret} must be a sub-type of C_x as the return value is assigned to x.

3.4.3 Finding Associated Code Elements

Pruning some code elements from an app requires all corresponding code elements to be pruned as well. For example, to remove an `Activity` from an app, all the inter-component communication links (e.g., Intent) start from and sink in this `Activity` must be pruned. To simplify our description,

```
1  class B extends A{
2      a(){t=new BgTask (this); trigger(t);}
3      b(){textView0 = findViewById(id0);
4          textView0.setText(data);}}
5  class C extends A{
6      c() {g = new BgTask(this);trigger(g);}
7      d() {textView1 = findViewById(id1);
8           textView1.setText(data.split(":")[0];
9           textView2 = findViewById(id2);
10          textView2.setText(data.split(":")[1];}}
11 class BgTask{ A ck; BgTask (A a) {ck=a;}
12  void run() {ck.data = getInputData();}}
```

Fig. 9: Demo code for discussion

we define the term "annotated *" (e.g., annotated statement) to represent the code annotated by feature location approaches. Term "tainted *" represents the code found to be related to the *annotated* code concerned. To locate the corresponding code elements to be pruned, we propose the following steps:

Pruning a type/class. To prune the type C from a program, we must eliminate all usages of type $C' <: C$. The C' represents the sub-type of C. The usage of type C' contains three parts: (T1) statements that involves the variables/fields in type C' (e.g., x =**new** $C(t)$); (T2) methods that return type C' or use parameters in type C'; (T3) the class of type C'.

Example. In Fig. 9, if the class A is pruned, the classes B and C must be pruned as well (see T3). The class $BgTask$ must be removed as well as the constructor that leverages the variable a whose type is A (see T1).

Pruning a method. To prune the method M from a program, we must prune all call sites of the method M. The call sites of the method M can be obtained from the PDG.

Pruning a statement, a variable, or a field. If the pruning target is a statement, a variable, or a field, we leverage both forward-data tracking and backward-data tracking to explore corresponding code elements to prune.

▷ **Forward-data Tracking.** The feature location approaches mark the programming elements to be pruned. These programming elements can be some variables or statements. Once these variables are pruned, their usages must be removed.

The *forward-data tracking* targets at finding all use of specific variables/fields. For example, the initialization expression $x := v$ defines the variable x. To remove this expression from the program, all uses of x following this expression must be pruned as well. To extract all uses, the *forward-data tracking* is applied. To perform the *forward-data tracking* on variables concerned (e.g., a variable definition statement to be pruned), we first annotate variables defined. Then, we forwardly annotate all variables by tracking the data flow on the PDG in an intra-procedure manner. This means we do not propagate the tracking procedure cross methods.

Example. In the Fig. 9, to explore all variables that access the variable $id0$ in Line3, we perform *forward-data tracking* to find the related variable $textView0$ in Line 3. The Line 4 is also considered, as it accesses the variable $textView0$.

▷ **Backward-data Tracking**. If some variables store some data (e.g. $x = v$ and x is an alias of v and x keeps a copy of v), we have to perform *backward-data tracking* to obtain the original of the data. We backwardly track the generation of the data used and all involved statements. If a variable x is an *annotated* variable and it is used as a parameter in a method call, the corresponding return value r should be *tainted*. Similar to *forward data tracking*, the *backward data tracking* backwardly traverses the PDG to find variables/statements can be *tainted*.

Example. In the Fig. 9, if the variable $textView2$ in Line 10 is an *annotated* variable, the *backward-data tracking* considers the variable $textView2$ and id2 are defined in Line 9 as *tainted*.

3.4.4 Building Specification

To wrap up, building specification of an app contains the following steps:

Step 1. XDebloat first initializes an empty *specification*, which contains all programming elements. The *decision*s of these programming elements are all set to the default (i.e., *unknown*);

Step 2. XDebloat reads configuration and feature location results to determine the programming elements to be pruned. Once finished, XDebloat changes the *decision* of them (ones to be debloated) to *remove*;

Step 3. XDebloat leverages the aforementioned approaches and rules to locate all other programming elements to be pruned. The decisions of programming elements explored in this step are set to *remove*;

Step 4. Lastly, XDebloat sets all *unknown* decisions to *preserve*; and

Step 5. XDebloat inspects the specification built with the type system (Sec. 3.4.2) and updates the specification when necessary.

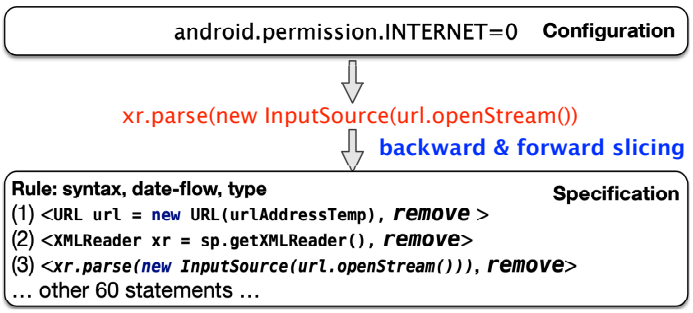

Fig. 10: Example of the configuration and specification for arxiv app.

Example. Recall the motivating example in Sec. 2.3, if developers intend to remove the feature *INTERNET* permission identified by the permission-based method, as shown in Fig. 10, developers need to set the $config$ of this feature to 0 in the configuration file. XDebloat automatically determines the programming elements associated with this feature. In this example, only one record "xr.parse(...)" is found, and then XDebloat performs data flow tracking to find all programming elements that need to be pruned. To determine whether a programming element can be debloated, XDebloat checks it through three rules. If so, XDebloat sets the *decision* of this element to *preserve*. Otherwise, the element's *decision* is set to *remove*.

In this example, XDebloat performs backward and forward data tracking on variable `xr` and `url` to find other statements that affect or are affected by these variables. XDebloat eventually finds 63 programming elements and checks whether they can be removed from the app. Finally, XDebloat generates the corresponding *specification*.

3.5 Debloating

XDebloat supports pruning-based and module-based debloating strategies. The latter reuses the techniques in the former to build individual modules.

3.5.1 Pruning-based Debloating

The pruning-based debloating contains the following steps:
Step 1. (Feature Location) Finding the features in the app with the feature location approach. This step is achieved by feature location approaches (Sec. 3.3);
Step 2. (Configuration) Users are then required to configure which features in the app should be removed. Once unwanted features are selected, XDebloat computes all related programming elements to prune;
Step 3. (Specification) For pruning-based debloating, once developers set up the features to be debloated in Step 2. XDebloate can automatically translate the configuration generated (in Step 2) to a *specification*. First, XDebloat initializes a *specification* that contains all programming elements concerned (i.e., code annotated in Step 1). Second, XDebloat sets *decisions* of all unwanted programming elements to *remove*. This is based on the *configuration* in Step 2. Third, we apply the rules for finding related code elements (Sec. 3.4.3) to obtain other programming elements that are needed to be pruned as well. When the rules (Sec. 3.4.3) are applied, the states of some programming elements are determined as *remove* or *preserve*; and
Step 4. (Debloating) In Step 3, the *specification* that contains all programming elements to be pruned in the app is built. In this step, XDebloat prunes programming elements based on the specification and outputs the target app. As XDebloat is built upon FlowDroid, this step can be achieved at the abstract syntax tree (AST) level. The debloated app generated XDebloat, cannot run on the device as it is repackaged. Thus, we remove the existing signature from the original app and sign the app again with our signature. If a component (e.g., Activity) is pruned from the app, we also remove the declaration of the component in the `AndroidManifest.xml`.

3.5.2 Handling Native Code

Apps can invoke the functions defined in the native libraries (`.so` libs) through Java Native Interfaces (JNIs). Instead of splitting the native code, we adopt a conservative strategy for handling native code. For a native library, if it is not used in any Java method, we can safely remove the native library. Otherwise, we preserve the library. We leave native code pruning to future work. Note that the APIs to load libraries (e.g.,`System.loadLibrary()`, `System.load()`) requires the existence of the target libraries. Such APIs are considered as usages of the library. For this case, we also preserve the library.

3.5.3 Handling Dead Code

Sometimes, dead code can be generated during the pruning process. For example, when all call sites of a callee are removed, the callee method becomes dead code. Therefore, we implement a dead code eliminator that can locate and remove dead code from the app. The Soot offers a dead code eliminator termed `DeadCodeEliminator`, our implementation is based on this API.

3.5.4 Removing ABIs and DPIs

Developers can have multiple sets of ABIs to support devices with different CPUs. Like RedDroid, XDebloat also supports pruning multiple sets of embedded ABIs in an app. In XDebloat, we support four types of CPUs: x86, x86_64, arm64_v8a, and armeabi_v7a. XDebloat uses Unix shell scripts to implement the redundant ABI remover. XDebloat first unzips the apk and decodes resource files with Apktool [43]. Based on users' configurations, XDebloat can remove unwanted ABIs and repackage the whole package into an apk. Similarly, an app can also hold Images with different Pixel Densities (DPI) for different types of devices. XDebloat can also prune multiple sets of DPIs in an app. In XDebloat, we support pruning ldpi (low-density), mdpi (medium-density), hdpi (high-density), and xhdpi (extra-high-density). XDebloat prunes multiple sets of DPIs with an AAPT command (`aapt r <path-to-files>`).

3.6 Module-based Debloating

3.6.1 Module-based Debloating with instant app

Since both the base module and feature modules in an instant app must contain at least one `Activity`, we use `Activity`-based feature location method to identify features in the original app. The intuition is that each module in the instant app must contain at least one `Activity` to enable users to access the `Activity` from URIs [12]. For all our feature location approaches, only the `Activity`-based feature location approach is qualified for this requirement. This is because the `Activity`-based feature location approach ensures each module (no matter a base feature module or a feature module) contains one `Activity`.
Requirements for instant apps. The instant app generated by XDebloat must fulfill the requirements set by Google[12]:
• Only nine permissions are allowed in an instant app. The supported permissions can be found in [12].
• The size of a feature module must be $\leq$ 4MB, and that of the base feature module must be $\leq$ 10MB.
• The target SDK version of instant apps must be $\geq$ 21.
Constructing instant app. Alg. 1 shows the process of building an instant app, which contains the following steps:
• Line 1-9: For the feature contains the main `Activity`, we build a base module for this feature. For other features, we build a feature module for each.
• Line 10-12: XDebloat fixes communications between different activities. This is because one module can communicate with anther only by specifying the URL in the `Intent` (Sec. 2). Therefore, we have to fix the original `Intent` with URLs. Note that XDebloat assigns a fixed URL starts with example.com (by default) to each module when building modules. For example, if URL example.com/a/ and example.com/b/ are assigned to

ALGORITHM 1: Constructing Instant App

```
Input  : P – a given app
Output: IP'– an instant app
1  P' ← annotate P with Activity-based feature location approach
2  for each Activity in P' do
3      if Activity is a main Activity then
4          Use Alg.2 to build the base feature module with Activity;
5      end
6      else
7          Use Alg.2 to build a feature module with Activity;
8      end
9  end
10 for each feature module do
11     Fixing inter-component communications by replacing Activties with URLs;
12 end
13 Move all assets and resources into base module
14 for each asset or resource file re do
15     if re is used by only a feature module f then
16         Move re to f
17     end
18 end
19 IP' ← build an instant app
20 Check whether IP' complies with the requirements for an instant app
```

Activity A and Activity B respectively, Activity A can navigate to Activity B by specifying the class name in the Intent (e.g, new Intent(this, ActivityB.class)). However, in an instant app, such communication can only be accomplished with Intents (by specifying the target URL) rather than using the target class name (e.g., new Intent(Intent.ACTION_VIEW,Uri.parse("example.com/b/"))). Therefore, XDebloat fixes such communications between components.

• Line 13-18: We work with resources (e.g., layout files) and assets (e.g., video, audio) while building the instant apps and app bundles. A conservative approach is putting all resources and assets in the base module. The reason is that for instant apps and app bundles, all feature modules can access the code, resources, assets in the base module without additional settings. However, putting all assets and resources in the base module can increase the size of the base module and violate the goal of debloating. Therefore, if we find a resource file re is only accessed by one feature module f, we move re to f. In this way, we remove resources/assets that are used by other feature modules from the base module.

• Line 19-20: An instant app is produced. Then, XDebloat checks whether the instant app built complies with the general requirements for an instant app.

Since module-base debloating reorganizes the app into different modules, we use DexPrinter (soot.toDex.DexPrinter) to build modules, because by default FlowDroid packages all classes in one apk file. Since in an instant app, each module can be built to an apk, we use DexPrinter to build each module with selected classes.

3.6.2 Module-based Debloating with App Bundle

Requirements for app bundle. The app bundle generated by XDebloat satisfies the requirements set by Google.

• The module size should be $\leq$ 150 MB.

• The target SDK version of an app bundle must be $\geq$ 21.

ALGORITHM 2: App Bundle Debloating

```
Input  : P – a given app
Output: AB– an App Bundle
1  P' ← identify features in P with Activity-based feature location approach
2  for each feature f in P' do
3      if feature contains the main Activity then
4          Build a base feature module for f
5      end
6      else
7          Build a feature module for f
8      end
9  end
10 Move all assets and resources into base module
11 for each asset or resource file re do
12     if re is used by only one feature module f then
13         Move re to f
14     end
15 end
16 Revise the code snippets that control the interactions between modules
17 Build the Module Protocol Buffer (*.pb) files and inject to each module
18 AB ← build/package the app bundle
19 Check whether AB complies with the requirements for an app bundle
```

Constructing app bundle. Alg. 2 shows the process of building an app bundle with three steps:

• Line 1-9: XDebloat assigns a unique module for each feature. If the feature contains the main Activity, this feature is considered as the base module. Otherwise, it is used to build a feature module;

• Line 10: XDebloat fixes all communications between modules. In an app bundle, the communications between different modules can be implemented with two steps: first, download the corresponding modules; second, access the component (e.g., Activity) in the target module directly. For the first step, the API SplitInstallManager.startInstall(SplitInstallRequst request) is used for installing the target module. For the second step, once the target module is installed, the component in the target module can be directly accessed with an Intent (e.g., new Intent().setClassName(getPackageName(),<tarclsName>), Note that tarclsName represents an explicit package name and a class name);

• Line 11-15: As aforementioned, we first temporarily put all assets and resources in the base module. Then, we check each asset and resource, if one resource file (asset) re is only accessed by one feature module f, we move re to f;

• Line 16-17: XDebloat builds the Module Protocol Buffer (*.pb) files [44] for each module and then injects them into each module. Module Protocol Buffer files provide metadata that describes the contents of each module. There are three type pb files: BundleConfig.pb presents about the information on the bundle; native.pb and resources.pb give the metadata on code and resources of the module. We build these files with Google's bundletool [45] and Android Asset Packaging Tool 2 (AAPT2) [46]. For building the BundleConfig.pb, the API AppBundle (in bundletool) is used. To build resources.pb file, we leverage the AAPT2 to

encode the resource files to `.flat` formats and package them to a `resources.pb` file. The `native.pb` is built with Google's bundletool as well; and

• Line 18-19: An app bundle is built. Then, XDebloat checks whether the app bundle built violates the requirements for an app bundle.

Example. Recall the motivating example in Sec. 2.3, to convert the arxiv app into an instant app, as an example, XDebloat generates one base feature module with 3 activities (i.e., `arXiv`, `RSSListWindow`, `PrintDialogActivity`), and 5 feature modules hosting `HistoryWindow`, `EditPreferences`, `SearchWindow`, `SearchListWindow`, and `SingleItemWindow`, respectively.

To create an app bundle, XDebloat builds a module for each feature, and the module that contains the main `Activity` (*arXiv*) is deemed as the base module. All other features are built as dynamic feature modules, which are downloaded from Google Play on-demand. For example, users first download and install the base module. If they want to search an article from the app, the feature module contains *SearchWindow* `Activity` is downloaded and executed.

3.7 Error Reporting in XDebloat

To prevent any unexpected errors introduced by debloating, XDebloat places a *error reporting* module for capturing crashes at runtime.

In general, there are two possible solutions to this. The first approach is instrumenting the debloated app with application performance management tools [47], [48]. The second approach is setting a crash handler for the entire app. In XDebloat, we choose the second approach. To be exact, we first check whether the app contains the instance of the `android.app.Application` class (`Application` for short). In Android, the `Application` class is the base class, which maintains the global application state [21]. If the app does not own the `Application` instance, XDebloat instruments one. Second, in the `Application` instance, XDebloat inserts a crash reporter in the `onCreate` method. It means when the app is initialized, our crash reporter is started. Last, the crash reporter we built is a subclass of `Thread.UncaughtExceptionHandler`. The `Thread.UncaughtExceptionHandler` can capture all uncaught exceptions in the app. Therefore, any unexpected crash can be captured by our crash reporter.

With the *error reporting* module, developers can directly leverage XDebloat to build and distribute the debloated app safely. The main purpose of adopting the error handler is to prevent runtime crashes and collecting runtime stack traces. There are two possible cases that can lead to runtime crashes: (1) there is a bug in the original app; and (2) a bug is introduced during debloating. Therefore, by implementing an error handler, we can determine the root cause (i.e., original app or debloating) of the bug. It also assists us in debugging our XDebloat. The error handler is a function of XDebloat for debugging purposes. But it is not necessary for the debloating tasks.

4 Evaluation

4.1 Experiment Setting

To evaluate the feature-oriented debloating approach proposed, we follow the evaluation methods used in the related debloating works [6], [7], [5], [9], [10]. Specifically, our evaluation is driven by the following research questions:

- RQ1 **Can the debloat apps be executed successfully?** Since XDebloat generates new apps, we first need to evaluate whether the debloated apps can be properly executed.
- RQ2 **Can XDebloat successfully turn the apps into instant apps and app bundles?** Since XDebloat supports *module-based* debloating, in this RQ, we intend to evaluate whether XDebloat can successfully turn apps into instant apps and app bundles.
- RQ3 **How much code can be removed by XDebloat?** We intend to show the effectiveness of XDebloat in removing bloated code. Here, we use the metric *debloating rate* defined in existing works to evaluate how much code could be removed by XDebloat.
- RQ4 **Does XDebloat contribute to resource (e.g, data, battery) reduction at runtime?** Removing code can contribute to resource reduction (e.g., data usage, battery usage), in this RQ, we evaluate resource reduction introduced by debloating.
- RQ5 **What is the execution time of XDebloat?** In this RQ, we target at evaluating the execution time of XDebloat.

To answer these research questions, we crawl 200 apps from F-Droid [49] and 1,000 real-world apps across different categories on Google Play. The settings for each research question are defined separately in this RQ.

Comparing with state-of-art approach. There are two existing works [6], [1] that are related to our work. However, as both works do not release their tools, we cannot make a comparison experimentally. Here, we only compare XDebloat with these two works in terms of functionality. RedDroid [6] offers two main features: (1) pruning multiple sets of ABIs; and (2) pruning dead code. These two features are also supported in XDebloat. Furthermore, XDebloat provides two debloating strategies, including pruning-based debloating and module-based debloating. Our work is different from Huang's work [1] in the following aspects: (1) the work [1] aims at pruning UI elements and their corresponding dependencies, such as pruning a button from an app. However, in our work, we do not consider any UI element (e.g, button, textedit, label) as a feature. Therefore, our work has different views on what to prune and what are the features comparing with [1]; and (3) our work supports two types of debloating: pruning-based and module-based debloating. Work [1] does not support module-based debloating.

Environment. All apps generated by XDebloat were tested on a Google Pixel smartphone running Android 8.1, which has 4GB RAM and 32GB storage.

4.2 RQ1: Can the debloat apps be executed successfully?

Motivation. Since XDebloat generates new apps, we need to test whether these apps can be executed properly.

Configuration. To test whether a debloated app can be executed correctly, we randomly generate a configuration for debloating. For example, if a feature location approach annotates 3 features in an app, we can have a random configuration $< 0, 1, 0 >$ to represent that we intend to prune feature f_1, f_3 and preserve feature f_2. The experiment is conducted on 200 open source apps.

Random configurations: The reason for using random configurations instead of testing all configurations is two-fold: a) with more features, the amount of possible configurations increases dramatically. For example, if we have 10 features, in total, there are 2^{10} possible configurations (i.e., 2^{10} possible apps); b) fuzzing all these possible options (2^{10}) requires a huge amount of efforts. For example, if we fuzz one app for 2 hours, in total, it requires 2^{11} hours to finish the task;

Open-source apps: We use open source apps in this RQ due to two reasons: a) The goal of this research question is to evaluate and assess the correctness of XDebloat, therefore, open source apps can be qualified candidates. We can easily instrument the apps with custom logs to evaluate our XDebloat framework (e.g, how XDebloat works, debugging XDebloat). For example, we can debug the debloated app and explore potential drawbacks in XDebloat. However, for commercial apps, such a goal can be hard to accomplish as most commercial apps are obfuscated and even hardened; and b) The fuzzer (i.e., Spanizer) we adopted relies on EMMA to collect coverage data. The EMMA tool instrument an app with the source code. Thus, we need the open-source apps for this task.

Number of apps: In this RQ, we use 200 open-source apps for evaluating XDebloat. We generate one random configuration for each feature location approach and thus we have 200 apps for each feature location approach. In total, we need to test 600 apps for all three feature location approaches. As a result, it costs our 1200 hours (50 days for a single machine) to test these apps as we fuzz each app for 2 hours.

Random configurations generated: The randomly generated configurations can be found on the project website for reference. As aforementioned, the motivation for adopting random configurations is based on the expected efforts of fuzzing. For a single app, it has 10 features, the number of configurations can be 2^{10}, the number of debloated apps can be $3{*}2^{10}$ (3 feature location approaches). Thus, fuzzing these apps requires $6 * 2^{10}$ hours (2 hours for each app), which is impractical.

Approach. We evaluate the debloated apps with an automatic testing tool named Sapienz [50]. Sapienz is a search-based testing tool. Sapienz initializes the initial population via MotifCore's Test Generator. Then, Sapienz performs a genetic evolution process, in which Chromosomes (test cases) are generated. Each time, Sapienz inputs a series of test events (e.g., click) to the app under test and collects the responses from the app under test. In Sapienz, a test case is defined as a series of test events. In Sapienz, a test case is passed if the test case is executed successfully; otherwise, it is considered as a fail. To obtain test coverage, we have to use the EMMA tool [51] to instrument the app. As a result, all crashes, coverage data, and test cases are stored in files by Sapienz. The motivation for using fuzzers is to evaluate whether XDebloat introduces errors during debloating. A higher test coverage indicates the most part of the subject app is tested. It also suggests that XDebloat's robustness in debloating. This is the reason that we leverage Sapienz and run the fuzzer for a long time (i.e., 2 hours) for each app to reach a higher test coverage. But, the exact test coverage data is not what we care about.

Our evaluation contains two steps.

Step 1: We instrument the debloated app with EMMA, and then run the app with Sapienz for 2 hours. Note that we reuse the seeds in Step 1 to ensure the fuzzing process keeps unchanged for the original app and debloated app.

Step 2: When the fuzzer (i.e. Sapnize) triggers a crash in a debloated app, a record is preserved for the purpose of replaying the crash. We can check if the original app also crashes for the same actions. Then, we can determine whether it is a bug from the original app. We skip all bugs that exist in both the original app and the debloated app because these bugs are not introduced by XDebloat.

Results: We run the Sapienz tool on each original app for 2 hours. Then, we manually inspect the bug report generated by Sapienz. If Sapienz reports a bug on the original app and it also reports the same bug on the debloated app, we consider the bug comes from the original app rather than introducing by XDebloat.

(1) `Activity`-base feature location. We randomly generate 200 configurations (one configuration for each app), and all 1615 test cases are passed. The fuzzing framework Sapienz can automatically generate test cases for the app under test. Sapienz generates a file for each test case while testing the target app. The number of generated files is counted as the number of test cases.

(2) Permission-based feature location. We randomly generate 172 configurations (one configuration for each app, 18 apps do not use any permission) and test them with 1415 test cases. Only 25 test cases failed, which are captured by our error reporter (see 3.7). After examining them, we found two reasons:

- The first one is due to XDebloat lacks of dynamic string analysis, which makes XDebloat fails to determine the callee.
- The second one is caused by some development frameworks in the apps, which bind code dynamically and thus XDebloat fails to capture such binding. For example, the app `com.google.android.stardroid` uses the development framework `javax.inject` for dynamic code binding. As using this framework may bypass the type checking system of XDebloat, XDeloat incorrectly treats some code fragments as dead code and prunes them.

Soundness of XDebloat. The root causes of the aforementioned failure cases are due to some dynamically generated values that cannot be resolved by XDebloat. Therefore, the soundness of XDebloat is limited by it does not support dynamic analysis at this stage.

(3) Modularity-based feature location. We randomly generate 200 configurations. 1972 out of 2023 test cases are passed. The errors are due to the same limitations presented above.

Manual Evaluation. Furthermore, we manually evaluate the debloated app to assess the correctness of XDebloat. Considering the workload, we randomly select 50 apps for evaluation. For each app, we first annotate the app with three different feature location approaches. Next, for each

feature location result, we randomly assign a configuration for debloating. For a debloated app, we build the window transaction graph (WTG) for the app to model the behavior of the app. A window represents a UI state in the app. The nodes in the WTG represent the windows in the app, while the edges represent the transactions between windows. The WTG suggests the operations should be performed to move from one window to another. We traverse the debloated app by referencing the WTG of the app. We record actions taken, the UI state of each window, and data displayed on the window by taking screenshots. The same procedure is performed on the original app. We compare the screenshots to determine whether functional correctness is preserved during debloating. In practice, the WTG is built with the state-of-art tool named GATOR [52]. By evaluating these debloated apps, we find that the functional correctness of the debloated apps is preserved.

4.3 RQ2: Can XDebloat successfully turn the apps into instant apps and app bundles?

Motivation. As XDebloat converts the original apps to instant apps and app bundles, we test whether they can be executed properly.

Approach. We further verify the generated app bundles using bundletool, a tool provided by Google for app bundle verification.

Results.

Results from instant apps. For the 200 apps, XDebloat can successfully transform all apps into instant apps. Among them, 164 instant apps can be published on Google Play without further modification. Other 36 apps require additional refactoring and reengineering to satisfy the requirements of instant apps. More precisely, among these 36 apps, 26 apps contain unsupported permissions. The remaining 10 apps are mainly due to unsupported or deprecated APIs for Android API 21.

By inspecting all apps which require human efforts, we find two cases that XDebloat cannot process automatically: (1) Unsupported or deprecated APIs for Android API 21+: Some apps use out-of-date APIs, and these APIs are no longer supported. Furthermore, to deploy an instant app, the target SDK for the app should be equal to or larger than 21; and (2) Unsupported permissions: As only certain permissions are allowed in instant apps, manual refactoring must be conducted on the app to adapt to the instant app framework.

Results from app bundles. For the 200 apps, XDebloat can successfully build 190 app bundles. Note that the generated App Bundle is in a specific format with the file extension `.aab`. It is possible to extract apks with help of the *bundletool* provided by Google. For the other 10 apps, they need extra reengineering as they used unsupported or deprecated APIs.

In summary, for most cases, the app bundles and instant apps built by XDebloat can be directly published to Google Play without extra human efforts. However, additional efforts from developers are required if an app uses unsupported API or permissions.

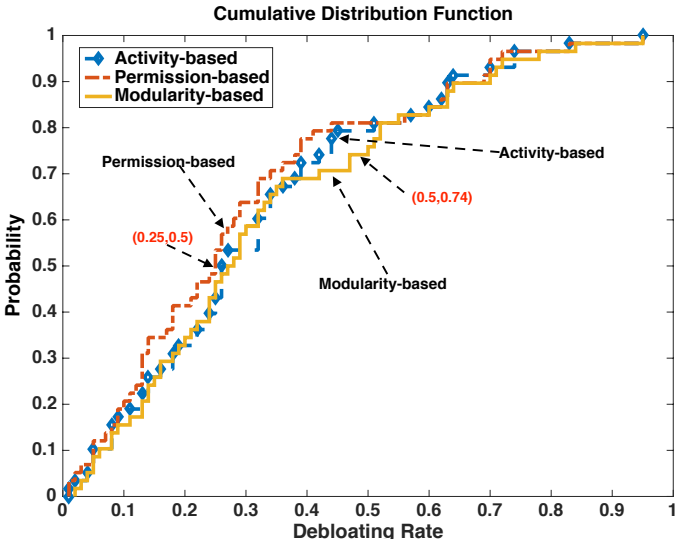

Fig. 11: Debloating rate for Conf1

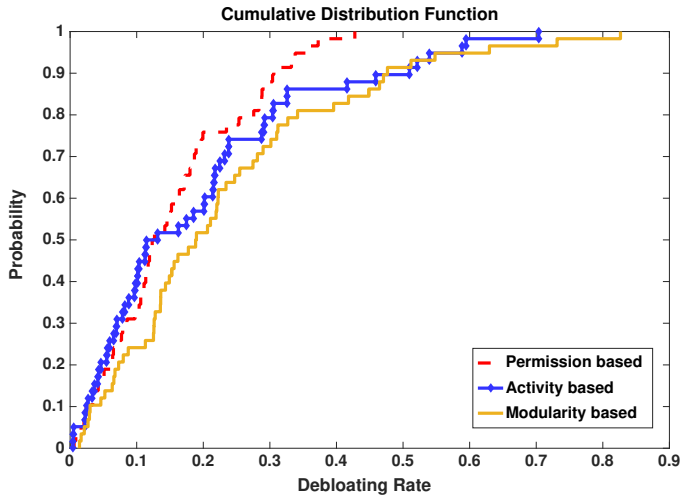

Fig. 12: Debloating rate for Conf2

4.4 RQ3: How much code can be removed by XDebloat?

Motivation. This RQ evaluates the effectiveness of XDebloat and the role that the feature location approaches play.

Configuration. Here, we propose two configuration strategies: in configuration 1 (**Conf1**), we disable all features annotated by feature location approaches. With this configuration, we can compute the maximal code fragments that can be pruned by XDebloat. In configuration 2 (**Conf2**), we only preserve the main features in the app and prune all other features to compute the debloating rate. For example, the media player feature is the main feature for a media play. For an email client app, sending and receive emails can be the main features. Other features can be considered as supporting features for the app. Base on this, we set up the configuration for each app, in which only main features are preserved and others are pruned.

Approach. We compute the debloating rate with the follow-

TABLE 2: Experimental Result for Removing all Features

App	Memory Usage							CPU Usage							Power Consumption						
	Ac			Per		Mod		Ac			Per		Mod		Ac			Per		Mod	
	O	D	R	D	R	D	R	O	D	R	D	R	D	R	O	D	R	D	R	D	R
VuDroid	1.52M	0.86M	43.2%	1.02M	32.6%	0.31M	79.8%	1m26s	1m10s	18.6%	56s	34.9%	34s	60.4%	10mAh	7mAh	30%	7mAh	30%	6mAh	40%
VLC	26.2M	22.4M	15.4%	20.20M	23.2%	13.8M	47.3%	14m24s	7m14s	49.7%	12m01s	16.5%	8m29s	41.0%	34mAh	25mAh	25.6%	19mAh	44.1%	22mAh	12%
AnkiDroid	11.20M	4.17M	62.8%	10.01M	10.6%	3.09M	72.4%	15m23s	3m23s	78.3%	6m16s	59.2%	1m23s	90.2%	23mAh	13mAh	43.4%	18mAh	21.7%	11mAh	52.2%
AudioMeter	1.67M	1.35M	19.1%	1.48M	12.0%	1.04M	37.4%	7m2s	6m43s	4.5%	6m14s	11.3%	3m23s	51.8%	3mAh	2mAh	33.3%	2mAh	33.3%	2mAh	33.3%
WeatherBug	16.84M	13.28M	21.1%	14.38M	14.5%	12.12M	28.2%	7m53s	7m4s	10.5%	6m41s	15.2%	6m27s	18.2%	7mAh	6mAh	14.2%	4mAh	42.8%	4mAh	42.8%
CityZen	10.8M	9.6M	11.1%	9.7M	10.1%	9.5M	12.0%	12m3s	9m53s	17.9%	9m20s	21.2%	7m41s	36.2%	23mAh	8mAh	65.2%	14mAh	39.1%	10mAh	56.5%
Delta Chat	25.0M	16.3M	34.7%	17.4M	30.2%	14.3M	42.8%	18m19s	15m20s	16.3%	15m14s	16.8%	13m37s	26.6%	45mAh	12mAh	73.3%	18mAh	60%	8mAh	86.6%
K9 Mail	8.37M	3.67M	56.2%	6.93M	17.2%	3.96M	52.7%	3m12s	2m5s	34.9%	2m38s	17.8%	2m6s	34.6%	12mAh	6mAh	50%	4mAh	66.6%	8mA	33.3%
Jamendo	416K	238K	42.7%	380K	8.6%	126K	69.7%	1m8s	50s	26%	1m1s	10.2%	48s	29.4%	4mAh	2mAh	50%	3mAh	25%	3mAh	25%
RedReader	4.84M	4.11M	15.0%	4.52M	6.61%	3.58M	26.2%	4m56s	4m10s	15.5%	4m34s	7.1%	2m59s	39.5%	9mAh	7mAh	22.2%	7mAh	22.2%	5mAh	44.4%

[1] Ac, Per, Mod: represent XDebloat leverages Activity-based, permission-based and modularity-based feature location approach, respectively; [2] O: original app, D: debloated app, R: ratio of change;

TABLE 3: Experimental Result for Preserving Main Feature

App	Memory Usage							CPU Usage							Power Consumption						
	Ac			Per		Mod		Ac			Per		Mod		Ac			Per		Mod	
	O	D	R	D	R	D	R	O	D	R	D	R	D	R	O	D	R	D	R	D	R
VuDroid	1.52M	1.32M	19.7%	1.32M	19.7%	1.09M	27.6%	1m26s	1m10s	18.6%	1m8s	16.3%	1m2s	27.9%	10mAh	8mAh	20%	8mAh	20%	6mAh	40%
VLC	26.2M	23.4M	10.6%	23.2M	11.4%	22.4M	14.5%	14m24s	13m4s	9.3%	13m1s	9.7%	12m29s	13.3%	34mAh	30mAh	11.7%	30mAh	11.7%	28mAh	17.6%
AnkiDroid	11.2M	9.1M	18.9%	10.01M	10.6%	7.1M	27.6%	15m23s	10m43s	30.3%	12m16s	20.2%	8m12s	46.7%	23mAh	17mAh	21.7%	18mAh	21.7%	17mAh	21.7%
AudioMeter	1.67M	1.51M	9.1%	1.53M	8.0%	1.45M	12.7%	7m2s	6m43s	4.5%	6m24s	9%	5m34s	20.8%	3mAh	2mAh	33.3%	2mAh	33.3%	2mAh	33.3%
WeatherBug	16.84M	16.08M	4.5%	16.45M	2.3%	15.98M	5.1%	7m53s	7m41s	2.3%	7m50s	0.6%	7m33s	14.2%	7mAh	7mAh	0%	7mAh	0%	5mAh	28.5%
CityZen	10.8M	9.6M	11.1%	9.7M	10.1%	9.5M	12.0%	12m3s	10m32s	12.5%	10m40s	11.2%	10m5s	16.2%	23mAh	19mAh	17.3%	21mAh	8.6%	17mAh	21.7%
Delta Chat	25.0M	21.5M	13.9%	22.5M	10.0%	21.9M	12.1%	18m19s	16m30s	9.28%	15m14s	16.8%	16m37s	9.27%	45mAh	38mAh	15.5%	42mAh	6.6%	38mAh	15.5%
K9 Mail	8.37M	7.91M	5.4%	7.8M	6.8%	7.51M	10.2%	3m12s	2m57s	12.8%	2m51s	10.8%	2m27s	23.6%	12mAh	9mAh	25%	10mAh	16.6%	8mA	33.3%
Jamendo	416K	397K	4.4%	395K	5.0%	380K	8.4%	1m8s	1m3s	7.5%	1m3s	7.5%	1m2s	8.8%	4mAh	3mAh	25%	3mAh	25%	3mAh	25%
RedReader	4.84M	4.31M	10.9%	4.62M	6.61%	4.22M	12.8%	4m56s	4m10s	15.5%	4m34s	7.1%	4m9s	15.6%	9mAh	7mAh	22.2%	9mAh	0%	8mAh	22.2%

[1] Ac, Per, Mod: represent XDebloat leverages Activity-based, permission-based and modularity-based feature location approach, respectively; [2] O: original app, D: debloated app, R: ratio of change;

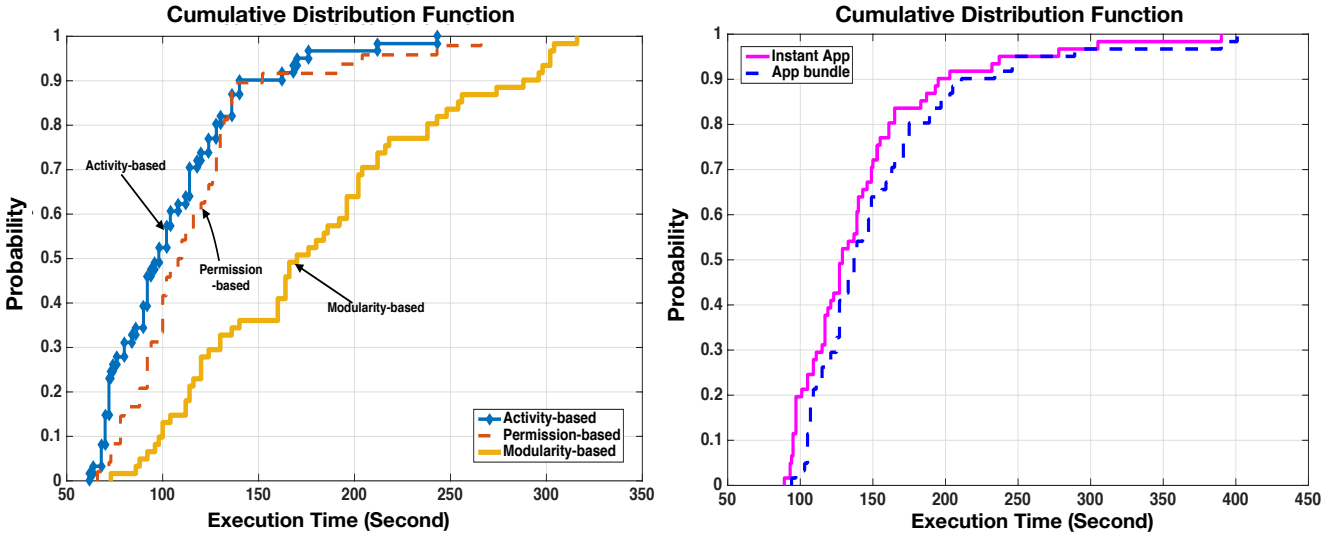

Fig. 13: Execution time of debloating

ing equation:

$$\text{debloating rate} = 1 - \frac{\text{size of debloated app}}{\text{size of original app}}.$$

A high debloating rate means that more components (features) in an app are pruned.

• Conf1: For Conf1, we evaluate the effectiveness of XDebloat on 1,000 commercial apps from Google Play and 200 open source apps.

• Conf2: For Conf2, we evaluate the XDebloat on 200 open source apps.

• **Results for Conf1.** As shown in Fig. 11, the average debloating rates of `Activity`-based feature location approach, permission-based feature location approach and modularity-based feature location approach are 34.33%, 29.31%, and 32.74%, respectively. Specifically, we find that for all three methods, 50% samples can reach a debloating rate higher than 25%. For all three approaches, at least, 26% of all samples can reach a debloating rate higher than 50%. Intuitively, the debloating rate of three feature location approaches can be differentiated from each other. However, from Fig. 11, there is no evidence to support this claim. We carefully inspect the reason. We find that this is mainly due to the strategy that we cope with native libraries (see Sec. 3.5.2), ABIs and DPIs (see Sec. 3.5.4). XDebloat adopts a conservative approach on the native libraries: if the debloated app does not use any JNI function, we removed the native libs in the app; otherwise we preserve the native libs. Furthermore, XDeloat also supports redundant ABIs and DPIs from an app. As a result, the debloating rates of all three approaches are similar since the libraries, images, and other resources share a large proportion of the apk. Removing libraries and resources can contribute much to the decrease in app sizes. But, XDebloat is not only a tool to shrink apps. XDebloat gives developers rights to determine what to be debloated by defining features. This is not supported in any other work.

• **Results for Conf2.** As shown in Fig. 12, the average

debloating rates of `Activity`-based feature location approach, permission-based feature location approach and modularity-based feature location approach are 19.12%, 15.38%, and 23.4%, respectively.

4.5 RQ4: Does XDebloat contribute to resources (e.g, data, battery) reduction at runtime?

Motivation. This RQ evaluates the reduction of the resources (e.g., data usage reduction, battery usage reduction) due to debloating.
Configuration. We use the same configuration as RQ3.
Approach. We randomly select 10 apps to manually compare and evaluate the effectiveness of debloating from memory usage, battery usage (a.k.a power usage), and CPU time. For each app, we use three feature location approaches to locate features and then leverage XDebloat to generate a debloated app. Thus, for each app, we can have three debloated apps. In total, we have 10 app groups, and each group contains one original app and three debloated apps. We run each app for 20 minutes on the Pixel phone to compute the memory usage, battery usage (a.k.a power usage), and CPU time. Specifically, to compute the memory and CPU usage of an app, we adopt the `adb` command "`adb shell dumpsys meminfo |grep <package-name>`" and "`ps -p <pid> -o TIME`", respectively. As for power consumption, we adopt the command "`adb shell dumpsys batterystats`" to compute it.
Result. Conf1. In this first configuration, we prune all features located by feature location approaches and then compute the resources reduction. Table 2 shows that Xdebloat can have average reduction rates of 32.1% (activity-based), 16.5% (permission-based), and 46.8% (modularity-based) in terms of memory usage. Xdebloat can have average reduction rates of 27.2% (activity-based), 21.0% (permission-based), and 42.7% (modularity-based) in terms of CPU usage. As for power consumption, the average performance of Xdebloat can be 40.7% (activity-based), 38.4% (permission-based), and 42.7% (modularity-based).

Conf2. In this configuration, we preserve the main feature and prune other features from the app to compute the resources reduction. Table 3 demonstrates that Xdebloat can have average reduction rates of 10.8% (activity-based), 9.1% (permission-based), and 14.3% (modularity-based) in terms of memory usage. Xdebloat can have average reduction rates of 12.2% (activity-based), 10.9% (permission-based), and 19.5% (modularity-based) in terms of CPU usage. As for power consumption, the average performance of Xdebloat can be 19.1% (activity-based), 14.3% (permission-based), and 25.8% (modularity-based).

Throughout the experiment, we find that pruning some features/functions from an app can lead to the reduction of resources in terms of CPU, memory, and even power. Some functions and features can be sensitive to computational resources, such as network connection (e.g.,WeatherBug), media play (e.g, VLC), GPS (e.g., CityZen). When these features are pruned, the pruned app may require fewer computation resources. We also find that it can be hard to claim what kind of feature requires more resources. It is because the amount of resources required for an app at runtime is determined by multiple factors, such as, how the feature is implemented.

4.6 RQ5. What is the maximal execution time of XDebloat?

Motivation. In this RQ, we intend to evaluate execution time of XDebloat.
Approach. To evaluate the execution time, we use 1000 commercial apps from Google Play and 200 open-source apps. First, we leverage three feature location approaches to annotate features in apps. Second, we prune all features from the apps to build the debloated apps. Third, we build instant apps and app bundles. The reason to prune all features is based on the intuition that pruning more features requires efforts (i.e., execution time) for XDebloat. Thus, by disabling all features, we can compute the maximal execution time of XDebloat. To the average execution time, for each app, we run every debloating approach ten times to reduce bias.
Results. Fig. 13 shows the cumulative distribution of the runtime overhead on average. Each point (x, y) in the diagram indicates that y of apps are debloated within x seconds. For example, point (150,0.6) represents 60% of all apps can be debloated in 150 seconds. From Fig. 13, we can conclude that : (1) the feature-location approach can influence the runtime performance; (2) the performance of activity-based and permission feature location approach is affected by the properties of the app (e.g., number of permissions declared, number of activity contained), whereas the performance of modularity-based feature location mainly depends on the structure of the app; (3) XDebloat usually requires more time for building app bundles comparing to building instant apps, because that building app bundles requires additional computational efforts for encoding resource files with module protocol (e.g., build `native.pb`, `resources.pb`). In general, XDebloat can accomplish a debloating task within 10 minutes.

5 Case Study

We further illustrate the applications of XDebloat with two cases.

5.1 Removing Unpopular/Unwanted Components in Apps

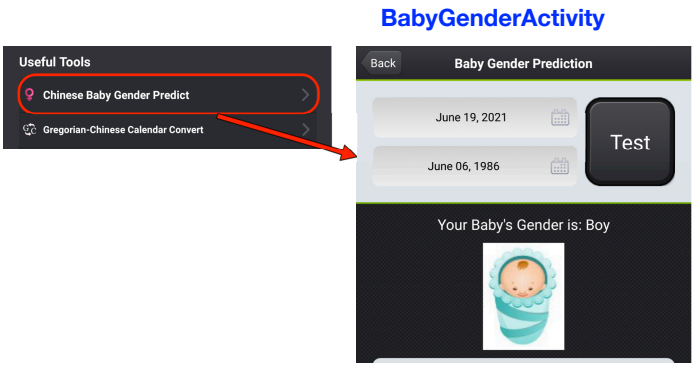

Fig. 14: Screenshots of China Calendar

There are several possible cases that developers want to prune a component from the app:

Case 1: if a feature in the app are not popular among end-users, developers can leverage XDebloat to remove the unpopular or unwanted feature from the apps to meet the market needs.

Case 2: if an app is out-sourcing to another company, it can be hard for the app owner to modify the app without the understanding of the app. The app owner can leverage XDebloat to modify the app.

The China Calendar app provides fundamental functions as a calendar. As shown in Fig. 14, the China Calendar app offers a *baby gender prediction* feature to predict the gender of a baby by referencing the conception date and user's birthday. If this feature is not popular among end users, developers can leverage XDebloat to prune the corresponding activity `BabyGenderActivity`. First, they can leverage XDebloat to annotate features with the `Activity`-based feature location approach. Second, we set up the `BabyGenderActivity` as the target feature to prune. Last, the corresponding code fragments that are associated with the `BabyGenderActivity` are pruned. The communications between `BabyGenderActivity` and other code fragments (see Fig. 15) are pruned as well.

```
1 switch(view.getId()){
2 case 2131034120: {
3     this.startActivity(new Intent(this,BabyGenderActivity.class));
4     break;
5  }
6  ...
7 }
```

Fig. 15: Communications between BabyGenderActivity

We further evaluate the correctness of the debloated app generated by XDebloat with the following steps: a) we leverage the dex2jar [53] tool to transform the debloated app to a jar; b) we manually check whether the unwanted code fragments have been removed from the code by reference the jar file. The jar file can be viewed with any Java decompiler (e.g, Luyten); c) we then check whether the component's declaration has been removed from the `AndroidManifest.xml`. After performing these checks, we find that XDebloat correctly pruned the component.

5.2 Removing Hardware-associated Features in Apps

RadarWeather app is a weather app, which offers weather forecasts for users. With the app, users can either set up the current city by searching the city name or set the city name with GPS. If locating cities with GPS is no longer supported, developers can leverage XDebloat to annotate the associated code with the permission-based feature location approach. As a result, there are two statements that are annotated with the GPS-related feature, including Line 3 in `updateLocation` method and Line 32 in `onUpdate` method in Fig. 16. Next, XDebloat explores other code fragments that should be pruned as well with backward-data tracking and forward-data tracking. Note that in Line 3, the variable `lastKnownLocation` is defined. With the backward-data and forward-data tracking, all uses of `lastKnownLocation` must be pruned. Therefore, XDebloat considers Line 4 to Line 15 should be pruned with the backward-data tracking and forward-data tracking.

Similarly, in `onUpdate` method, Line 22 to Line 31 are also considered as candidates to prune. By removing all code fragments that require location permission, we prune the GPS-related feature from the app.

We further evaluate the correctness of the debloated app generated by XDebloat with the same step in Sec. 5.1. We find that the corresponding code fragments have been removed correctly. We install the debloated app on a real device. We find that the debloated app does not require the location permission and does not set up the location with the GPS data.

6 Discussion

6.1 Feature Location Approaches

In this work, we present three feature location approaches, including `Activity`-based approach, permission-based approach, and modularity-based approach. Moreover, developers can leverage XDebloat to annotate features in an app with any granularity: class, method, statement, and field. In practice, since the features of an app should be only defined by its developers [29], we do not claim our feature location approaches as the oracle.

6.2 The Usage Scenario of Feature-oriented Debloating

The feature-oriented debloating approach presented in this paper can be applied to various scenarios. Specifically, we present several common cases that developers can apply XDebloat.

1) *Generate a lightweight version of a legacy app*: This is the primary usage and common practice of feature-oriented debloating. Developers can build an app with all features, which can be considered as the full version, to make a profit. Meanwhile, they can leverage XDebloat to build a lightweight one, in which certain features are included. As a marketing strategy, developers can publish the lightweight one for free and the full version at a certain price. If users are satisfied with the lightweight one, they can pay for the full version.

2) *Build Instant App and App Bundle from a legacy app*: Google promotes the instant app and app bundle as the future of Android development. With XDebloat, app developers can build instant apps and app bundles automatically from their legacy apps rather than building from scratch.

3) *Remove vulnerable components from an app*: If a vulnerable component in an app is exploited and developers cannot fix it immediately, one promising solution is removing the vulnerable component from the app. When developers prepare the patch, they can upload the patched one to app stores. In this case, developers can annotate vulnerable components with the GUI toolkit offered by XDebloat. XDebloat can then prune the vulnerable component from the app.

6.3 Limitations

The limitations in this study include: (1) being a static analysis-based tool, XDebloat has some inherent problems in the static analysis, such as it cannot handle some values that are dynamically generated at runtime. In future work,

```
public static void updateLocation(final Context context, final int n) {
 if (ContextCompat.checkSelfPermission(context, "android.permission.ACCESS_COARSE_LOCATION") == 0) {
  final Location lastKnownLocation = ((LocationManager)context.getSystemService("location")).getLastKnownLocation("gps");
  if (lastKnownLocation != null) {
    final double n2 = Math.round(lastKnownLocation.getLatitude() * 100.0) / 100.0;
    final double n3 = Math.round(lastKnownLocation.getLongitude() * 100.0) / 100.0;
    for (int i = 0; i < allCitiesToWatch.size(); ++i) {
        if (allCitiesToWatch.get(i).getCityId() == n) {
            final CityToWatch cityToWatch = allCitiesToWatch.get(i);
            cityToWatch.setLatitude((float)n2);
            cityToWatch.setLongitude((float)n3);
            break;
        }
    }
  }
  ...
  }
}

public void onUpdate(final Context context,...) {
 if (defaultSharedPreferences.getBoolean("pref_GPS", true) == Boolean.TRUE) {
  if (WeatherWidget.locationListenerGPS == null) {
    WeatherWidget.locationListenerGPS = (LocationListener)new LocationListener() {
     public void onLocationChanged(final Location location) {
        final int[] appWidgetIds = ...;
        for (int length = appWidgetIds.length, i = 0; i < length; ++i) {
            WeatherWidget.this.updateAppWidget(context, appWidgetIds[i]);
        }
     }
     ...
   };
  this.locationManager.requestLocationUpdates("gps", 300000L, 10000.0f, WeatherWidget.locationListenerGPS);
  }
 }
 ...
}
```

Fig. 16: GPS-related Code in RadarWeather

we will enhance XDebloat with the capability of conducting dynamic analysis; (2) currently, we use a conservative approach on the native libraries (Sec.3.5.2). We plan to support native code debloating in XDebloat in the future; (3) at this stage, XDebloat does not support apps that have been obfuscated. The main problem comes from the feature location part. If an app is obfuscated, the feature location may fail to locate features correctly, especially for the permission-based feature location approach, because this feature location approaches rely on the method signatures for locating features in an app.

6.4 Threats to Validity

Our study explores the feature-oriented debloating on real-world apps. Regarding the external validity, as the number of apps selected for evaluation is not large, the experimental results may be biased. Hence, we do not attempt to generalize our results to all apps. Regarding internal and construct validity, the measurements of debloating rate, execution time, and resource reduction can evaluate our approach. The experiments are conducted with real-world apps and experiments are performed on a real device, thus the internal and construct validity are met.

7 Related Work

This section introduces related studies on program debloating, bloat detection, program customization, and program slicing.

Program debloating. Program debloating aims at pruning unused functions in a program. Heo et al. [8] proposed a reinforcement learning guided debloating approach for C programs. Quach et al. [14] proposed a piece-wise debloating approach for pruning unused C functions at the compiling and loading phase, respectively. Furthermore, CIMPLIFIER allows users to debloat application containers (i.e., Docker) based on user-defined constraints [54]. TRIMMER leverages user-provided configuration data to trim a C application [5]. Besides the approaches for C programs, some recent studies aim at Java programs. JRed [7] is a static approach to trim unused redundant bytecode off the application and the runtime JRE. RedDroid prunes dead code in Android apps based on static analysis [6]. Different from these studies, XDebloat can not only prune unused functions but also turn the bloated apps into instant apps and app bundles. Moreover, it provides approaches to identify features.

Software bloat detection. Bhattacharya et al. [4] presented an approach to detect the source of software bloat in Java application. Xu et al. [3] and Bu et al. [55] discussed the influence and impact of software bloat. There is also a large body of research [56], [57] on detecting runtime memory bloat. Such studies are out of the scope of our work.

Program reduction and customization. Program reduction aims at generating the smallest variant of the original program while keeping the same property as the original one. Different from debloating, program customization approaches try to customize the program mainly for testing and debugging purposes.

Sun et al. [9] presented a syntax-guided program re-

duction approach for C. It yields a minimized program variant that holds the equivalent properties with the original one without introducing syntax errors. C-Reduce [10] employed a series of heuristics for program reduction based on semantics obtained from Clang. Different from these program reduction and customization approaches, we intend to customize an app from the perspective of features. Moreover, these approaches do not support features during customization.

Program slicing. Another widely adopted framework for program reduction is program slicing. Slicing is used for certain purposes. For example, debugging, testing, software customization, optimization [58], [59]. A slice is a subpart of the original program, and it respects the value at some points of interest. Different from these slicing works [58], [59], [60], feature-oriented debloating aims at pruning unwanted functions based on a user-defined configuration instead of specifying values of interest.

8 Conclusion

We propose a novel feature-oriented debloating approach and develop XDebloat to automate this process. XDebloat supports feature location methods at all granularities and can perform pruning-based and module-based debloating for apps. Besides removing unwanted features, it can also turn a bloated app into an instant app or an app bundle. We evaluate XDebloat with 1,200 real apps, and the experimental results show that XDebloat can successfully prune them or transform them into instant apps and app bundles in a few minutes.

References

[1] J. Huang, Y. Aafer, D. Perry, X. Zhang, and C. Tian, "Ui driven android application reduction," in *Proc. of ASE*, 2017, pp. 286–296.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines*. Springer, 2016.

[3] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proc. of the FSE/SDP Workshop*, 2010, pp. 421–426.

[4] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," in *Proc OOPSLA*, 2013, pp. 745–764.

[5] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: Application specialization for code debloating," in *Proc. of ASE*, 2018, pp. 329–339.

[6] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *Proc. of ISSRE*, 2018, pp. 189–199.

[7] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *Proc. of COMPSAC*, 2016, pp. 12–21.

[8] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proc. of CCS*, 2018, pp. 380–394.

[9] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proc. of ICSE*, 2018, pp. 361–371.

[10] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," *SIGPLAN Not.*, vol. 47, no. 6, pp. 335–346, 2012.

[11] Google, "App bundle," https://developer.android.com/platform/technology/app-bundle/, 2019.

[12] ——, "Instant app," https://developer.android.com/topic/google-play-instant/, 2019.

[13] I. Corporate, "Smartphone market share," https://www.idc.com/promo/smartphone-market-share/os, 2021.

[14] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proc. of USENIX Security*, pp. 869–886.

[15] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proc. of ICSE*, 2015, pp. 77–88.

[16] XDebloat, "Xdebloat online artefact," http://sites.google.com/view/xdebloat, 2021.

[17] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, "All your app links are belong to us: Understanding the threats of instant apps based attacks," in *Proc. of ESEC/FSE*, 2020, pp. 914–926.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. of PLDI*, 2014, pp. 259–269.

[19] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," in *Proc. of ASE*, 2013, pp. 652–657.

[20] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *Proc. of NDSS*, 2015, pp. 1–15.

[21] Google, "Android documentation," https://developer.android.com, 2021.

[22] A. S. Ami, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Systematic mutation-based evaluation of the soundness of security-focused android static analysis techniques," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, pp. 1–37, 2021.

[23] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection - literature review and empirical study," in *Proc. of ICSE*, 2017, pp. 507–518.

[24] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *Proc. of USENIX Security*, 2018, pp. 1669–1685.

[25] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proc. of USENIX*, 2013, pp. 543–558.

[26] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, p. 318–329.

[27] T. Johann, C. Stanik, A. M. A. B., and W. Maalej, "Safe: A simple approach for feature extraction from app descriptions and app reviews," in *Proc of RE*, 2017, pp. 21–30.

[28] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semiautomatic detection of product-line features," *IEEE Transactions on Software Engineering*, vol. 40, pp. 67–82, 2014.

[29] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: A qualitative study of features in industrial software product lines," in *Proc. of SPLC*, 2015, pp. 16–25.

[30] D. Nešić, J. Krüger, u. Stănciulescu, and T. Berger, "Principles of feature modeling," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, p. 62–73.

[31] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, p. 155–166.

[32] Android, "Device permissions that imply device hardware use," https://developer.android.com/guide/topics/manifest/uses-feature-element.html#permissions-features, 2021.

[33] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. of CCS*, 2012, pp. 217–228.

[34] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proc. of S&P*, 2016, pp. 636–654.

[35] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. of ICSE*, 2014, pp. 1025–1035.

[36] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security

vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, pp. 1–32, 2018.
[37] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 1–41, 2017.
[38] G. Nudelman, *Android design patterns: interaction design solutions for developers*. John Wiley & Sons, 2013.
[39] D. Lettner, K. Eder, P. Grünbacher, and H. Prähofer, "Feature modeling of two large-scale industrial software systems: Experiences and lessons learned," in *Proc. of MODELS*, 2015, pp. 386–395.
[40] B. Behringer, J. Palz, and T. Berger, "Peopl: Projectional editing of product lines," in *Proc. of ICSE*, 2017, pp. 563–574.
[41] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75 – 174, 2010.
[42] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE International Parallel and Distributed Processing Symposium*, 2018, pp. 885–895.
[43] C. T. C. M. R. W. O. Creator, "Apktool: A tool for reverse engineering android apk files," https://ibotpeaches.github.io/Apktool/, 2021.
[44] Google, "Protocal buffer," 2021. [Online]. Available: https://developers.google.com/protocol-buffers/
[45] ——, "bundletool," https://github.com/google/bundletool, 2021.
[46] Android, "Android asset packaging tool," 2021. [Online]. Available: https://developer.android.com/studio/command-line/aapt2
[47] Y. Tang, X. Zhan, H. Zhou, X. Luo, Z. Xu, Y. Zhou, and Q. Yan, "Demystifying application performance management libraries for android," in *Proc. of ASE*, 2019, pp. 682–685.
[48] Y. Tang, H. Wang, X. Zhan, X. Luo, Y. Zhou, H. Zhou, Q. Yan, Y. Sui, and J. W. Keung, "A systematical study on application performance management libraries for apps," *IEEE TSE*, pp. 1–20, 2021.
[49] F-Droid, "F-droid," https://f-droid.org, 2021.
[50] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. of ISSTA*, 2016, pp. 94–105.
[51] Emma, "Emma," htpp://emma.sourceforge.net/index.html, 2019.
[52] A. Rountev, "Gator: Program analysis toolkit for android," http://web.cse.ohio-state.edu/presto/software/gator/.
[53] Dex2Jar, "Dex2jar," https://github.com/pxb1988/dex2jar, 2021.
[54] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proc. of ESEC/FSE*, 2017, pp. 476–486.
[55] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," *SIGPLAN Not.*, vol. 48, no. 11, pp. 119–130, 2013.
[56] K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu, "Understanding and combating memory bloat in managed data-intensive systems," *ACM TOSME*, vol. 26, no. 4, pp. 12:1–12:41, 2018.
[57] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *Proc. of PLDI*, 2009, pp. 419–430.
[58] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *Proc. of ESEC/FSE*, 2017, pp. 174–185.
[59] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *Proc. of HASE*, 2016, pp. 122–131.
[60] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proc. of SAC*, 2013, pp. 1844–1851.