APPCOMMUNE: Automated Third-Party Libraries De-duplicating and Updating for Android Apps

Bodong Li, Yuanyuan Zhang, Juanru Li, Runhan Feng, Dawu Gu
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China
{uchihal, yyjess, jarod, fengrunhan, dwgu}@sjtu.edu.cn

***Abstract*—The increasing usage of third-party libraries in Android apps is double-edged, boosting the development but introducing extra code base and potential vulnerabilities. Unlike desktop operating systems, Android does not support the sharing of third-party libraries between different apps. Thus both the de-duplicating and the updating of those libraries are difficult to be managed in a unified way.**

In this paper, we propose a third-party library sharing method to address the issues of code bloating and obsolete code updating. Our approach separates all integrated third-party libraries from app code and makes them still accessible through a dynamic loading mechanism. The separated libraries are managed centrally and can be shared by different apps. This not only saves the storage but also guarantees a prompt update of outdated libraries for every app. We implement APPCOMMUNE, a novel app installation and execution infrastructure to support the proposed third-party library sharing without modifying the commodity Android system. Our experiments with 212 popular third-party libraries and 502 real-world Android apps demonstrate the feasibility and efficiency: all apps work stably with our library sharing model, and 11.1% storage and bandwidth are saved for app downloading and installation. In addition, APPCOMMUNE updates 86.4% of the managed third-party libraries (with 44.6% to the latest versions).

***Index Terms*—Android, Third-party Libraries, Code Updating**

I. INTRODUCTION

Mobile app development involves a wide variety of third-party libraries. Different apps often rely on a small set of popular third-party libraries to fulfill similar functions. Although leading mobile operating systems (Android and iOS) do inherit library sharing mechanisms from desktop operating systems (Linux and Mac OS), only a limited number of system provided libraries are shared and user apps could neither add nor update them. As a result, mobile apps have to statically integrate all used third-party libraries.

The individual integration of third-party library brings many issues to the mobile ecosystem. First, it is very difficult to update the used third-party libraries in all apps. Actually, according to the survey conducted by Erik *et al.* [1], developers fail to update a third-party library due to a variety of reasons such as incompatibility issue, update unawareness, or actively ignoring the new version. Thus a large portion of third-party libraries are outdated even if the host apps are updated to the latest version [2] [3] [1]. Second, the individually downloaded and stored third-party libraries consume a huge amount of both network traffic and local storage space. In fact, third-party library repetition is very common for popular mobile apps. If the duplicated third-party libraries are not repeatedly downloaded and stored, app markets and mobile devices can both benefit from this de-duplicating.

Existing studies of third-party library management on mobile platform often focus on the permission control. Several approaches [4] [5] [6] [7] are proposed to restrict the potential harmful behaviors and unnecessary permissions of third-party libraries. Other approaches [8] radically remove the malicious libraries from the repacked apps. Despite all those efforts, mobile platforms still lack an effective mechanism to both reduce library repetition and help the updating of obsolete libraries.

To help developers and mobile devices manage third-party libraries, in this paper we propose a library sharing strategy for Android apps. Our strategy helps Android apps share third-party libraries without the modification of the commodity Android system. To fulfill this, all third-party libraries are first separated from an app. Then the separated app and libraries are respectively downloaded and installed. To reduce library repetition and promote library updating, third-party libraries are stored and managed in a centralized way and hence apps do not need to update the used libraries individually.

We implement APPCOMMUNE, a novel app installation and execution infrastructure to support our third-party library sharing strategy. Multiple apps on the same device with the support of APPCOMMUNE would share the same third-party library, which saves both the bandwidth and the storage. APPCOMMUNE consists of a **Market Proxy** on the server side and a **Lib Manager** on the client side. The Market Proxy automatically separates all third-party libraries from an app to generate a tailored app, and then rewrites both the tailored app and third-party libraries for a later dynamic loading based execution. If a user wants to install this app, she will download the tailored version and install it. After that, the Lib Manager will automatically check and download necessary third-party libraries for this app.

APPCOMMUNE introduces novel app trimming and rewriting techniques to support library sharing without modifying the commodity Android system. This allows the tailored apps

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

to dynamically load the shared third-party libraries. Moreover, on the premise of stability, APPCOMMUNE automatically adapts the proper library version (often the new version) for the tailored app to promote the library updating and thus reduce security threat of the vulnerability in third-party libraries. To validate the effectiveness, we tested APPCOMMUNE with 212 popular third-party libraries and 502 real-world Android apps on 10 different Android devices. The result showed that all tailored apps worked normally under our library sharing model with almost no performance overhead. In addition, APPCOMMUNE saved 11.1% storage and bandwidth usage for the downloading and installing of those apps, and updated 86.4% of the used third-party libraries for the tested apps (44.6% to the latest version). With a vulnerability detection against five publicly released security vulnerabilities related to third-party libraries, APPCOMMUNE successfully protected all 31 influenced apps through the library updating.

In summary, this paper makes the following contributions:

- We propose a third-party library sharing strategy in Android system to tackle the bloated and obsolete third-party libraries. To the best of our knowledge, our solution is the first one to address the updating of outdated Android third-party libraries without modifying the system.
- We implement APPCOMMUNE, app installation and execution infrastructure to support our library sharing strategy. APPCOMMUNE introduces app trimming and rewriting techniques to help apps use shared third-party libraries, and conducts comprehensive updating test to adapt apps with proper versions of libraries.
- The experiments with popular Android apps and third-party libraries demonstrate that the deployment of APPCOMMUNE on commodity Android devices is expected to reduce the expense of data transmitting and storing, and enhance the security of the ecosystem.

II. MOTIVATION

A. Problems

Although third-party library simplifies the app development, it brings two issues: duplicate libraries lead to the bloating of app size, and outdated libraries contain potential vulnerabilities. Here we give a concrete example with two apps to describe these issues in detail. As Figure 1 shows, two apps (Noogra Nuts[1] and MAPS.ME[2]) contain 17 and 11 third-party libraries respectively. Among those integrated third-party libraries, six libraries are shared by both apps. If a user installs both those two apps on the same mobile device, these six libraries are downloaded repeatedly within different apps. Obviously, this consumes more network traffic and flash storage. Furthermore, we notice that most libraries of Noogra Nuts are in older versions compared with MAPS.ME. The use of outdated third-party library is insecure. For instance, the specific version of Facebook library (version 3.15) used in

[1]Noogra Nuts is a popular game app [9].

[2]MAPS.ME is an open-source offline maps app [10].

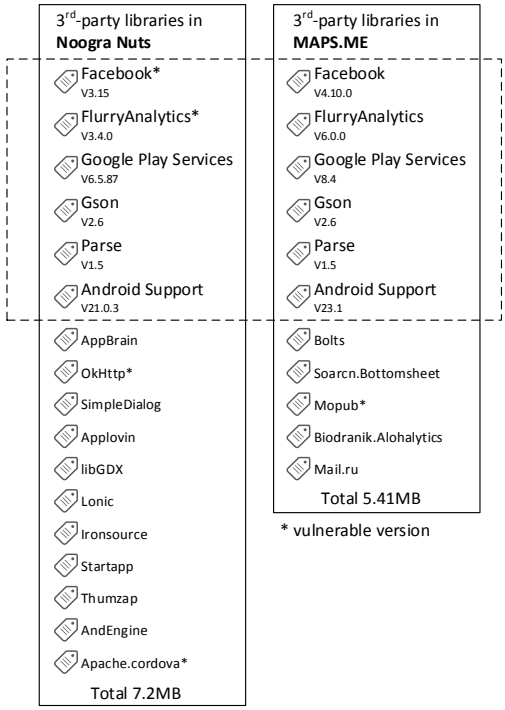

Fig. 1. Two apps contain six same third-party libraries, some libraries are even vulnerable.

Noogra Nuts contains at least three security vulnerabilities according to Facebook's official specifications [11].

B. Approach and Challenges

To address the issues of third-party library, we propose a third-party library sharing strategy to reduce library repetition and promote library updating. In general, our strategy separates all third-party libraries from an app before it is downloaded and installed. Then, the traditional app installation process is divided into two individual processes: an installation of a **tailored app** (app without any third-party libraries), and an installation of necessary third-party libraries. On the premise of stability, only a few (one or two) secure and stable versions of certain third-party library (often the latest version) is required to be installed on the device and shared by multiple apps. In this manner, third-party libraries are managed in a centralized way and two advantages–**library de-duplicating** and **library updating**, can be easily achieved.

Nevertheless, there exist many challenges for a stable and efficient library sharing mechanism:

How to separate third-party libraries from an app: Android system supports two kinds of libraries: java library (`.jar`) and native library (`.so`). Within an `apk` file, native libraries are stored independently in `/libs` directory, while the java libraries and the original app code are compiled together into a compact Dalvik Executable (`dex`) file. The challenge here is how to identify and split the integrated third-party java

```
public interface FacebookCallback {
    void onSuccess(Object arg1);
}
```

(a) declaration in library code

```
public class FacebookCallbackAdapter implements FacebookCallback {
    public void onSuccess(Object arg1) {
      ...
    }
}
```

(b) implementation in app code

Fig. 2. The figure shows an example of interface-typed API, which is not suitable for dynamic loading.

libraries from the `dex` file and still keep the code of main functionalities.

How to share third-party libraries between different apps: Originally, third-party libraries are not designed for app sharing, and an app also does not support the API invoking of shared third-party libraries outside its own sandbox. In order to enable such mechanism, a series of app and library rewriting measures must be provided. Since third-party libraries are removed from the original app, the API invoking of those libraries should adopt a new pattern. That requires the rewriting of all instructions related to third-party library API invoking. Moreover, app often utilizes a dynamic loading mechanism to invoke a shared library other than a statically integrated library. However, not all kinds of API invoking can be rewritten with a dynamic loading.

As shown in Figure 2, the interface-typed `FacebookCallback` API is implemented directly by `FacebookCallbackAdapter`. This interface must be initialized before the implementation, so it is not allowed to be loaded afterwards through a dynamic loading.

How to handle API changes among different versions of third-party libraries: The updating of third-party libraries should also concern about the API compatibility issue. If the new version of library changes too much compared to the old one, the updating may affect the normal functionalities of the app. So the updating not only should consider the accuracy of third-party library identification, but should also consider how to choose a proper version of the library to adapt the app.

C. Insights

To address the three challenges mentioned above, we propose the following key techniques to help implement third-party sharing on commodity Android devices:

Third-party Library Identifying and Separating: The separating of third-party libraries requires an exact identification, we need to identify each library and even its exact version. To implement this, we utilize class hierarchy as the core feature to identify used third-party libraries. It's independent of the detailed class names or method names, so we can effectively

```
FacebookSdk.sdkInitialize(context);
```

(a) direct invocation

```
DexClassLoader loader = new DexClassLoader(dexpath, ...);
class clz = loader.loadClass("com.facebook.FacebookSdk");
Method mth = clz.getDeclaredMethod("sdkInitialize",Context.class);
mth.invoke(clz,context);
```

(b) reflective invocation

Fig. 3. A comparison between two kinds of invocations

resist most layout obfuscations (which is the most common kind of code obfuscations in Android). Then we disassemble an app and utilize the collected features to match the integrated third-party libraries in it. After the identification, we remove all code of third-party libraries for a further **tailored app** rebuilding.

Rewriting of App and Third-party Library: In our third-party library sharing scenario, both (tailored) apps and third-party libraries are rewritten to adjust the execution model. As Figure 3 shows, the reflective invocation is much more complex than the direct invocation. Therefore, we do not directly change each API invocation into java reflective mode. Instead, we first add a wrapper for each API of a third-party library. Then we rewrite the app to add declarations of all wrappers into the app. In addition, we insert a wrapper initializer that dynamically loads modified third-party libraries and obtains the list of wrappers through java reflection before the execution of the app. Finally, we replace each invoking instruction of third-party library API with an invoking instruction of the corresponding wrapper. In this way, the patching of original invoking instructions is significantly simplified.

Specially, we would automatically check all reflection related methods in app code. If any API is originally invoked by java reflection, we can directly change the invocation target to corresponding wrapper. Therefore, original reflective invocation is not our obstacle.

Library Updating and Version Adapting: To guarantee the stability of app execution, here we adopt a conservative updating strategy. We divide the update of library into three types according to the study of Erik *et al.* [1]. A **major change** indicates that backwards incompatible API changes or additions are added. A **minor change** indicates that the changes or additions are backwards compatible. A **patch change** indicates that there are only bug fixes or code-only changes that do not affect APIs. For an app which contains an outdated library, only when the used APIs contain no major changes between the new version and the old one, the updating would be adapted. According to this principle, we try to update the third-party libraries to new versions. In this way, there would be more than one versions of certain library installed in the same device (according to the detailed installed

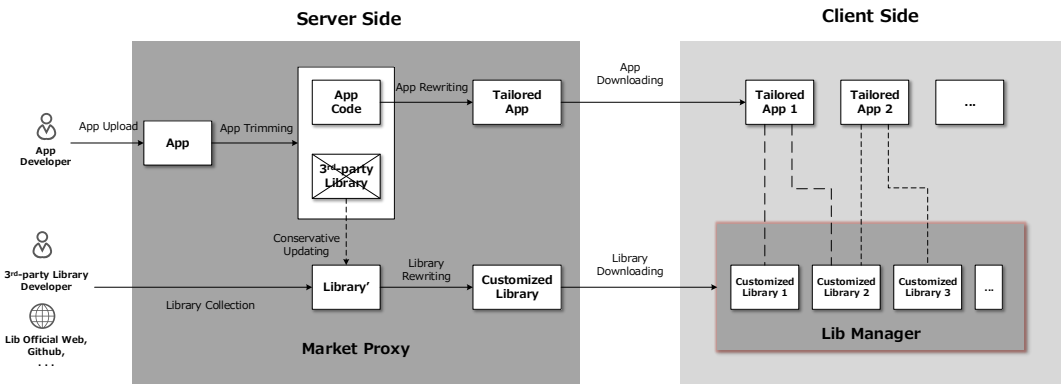

Fig. 4. An overview of APPCOMMUNE

apps). Some apps share the latest version and others share an older version. Although we don't maximize the library de-duplicating, it can also effectively save the space and ensure the stability simultaneously.

III. APPCOMMUNE

A. Overview

We propose APPCOMMUNE as a new Android app market model to achieve our third-party library sharing strategy. Figure 4 depicts the architecture of APPCOMMUNE. On the server side (i.e., app market) , APPCOMMUNE introduces a **Market Proxy** to trim an app and update the third-party libraries. The updated libraries and the tailored apps will be sent to devices of end-users respectively. On the client side (i.e., local device), APPCOMMUNE introduces a **Lib Manager** to manage third-party libraries in a centralized manner and stitches those tailored apps with libraries of proper version.

When APPCOMMUNE is deployed, it first collects most mainstream third-party library SDKs. These third-party libraries are collected from network resource or uploaded by third-party library developers. After the collection of libraries, APPCOMMUNE starts to serve as a normal app market. App developers can upload their apps to the server side. Moreover, APPCOMMUNE would requests app developers to help re-sign the tailored apps since it needs to repack each app on the server side.

On the server side, the Market Proxy performs three main tasks: 1) It trims all identified third-party libraries in an app and rebuilds the rest part as a tailored app; 2) According to the detailed used APIs, it determines whether a library should be updated and the proper version. 3) It patches the corresponding collected library with proper version to add a wrapper layer for tailored apps to invoke. As a result, the tailored apps and customized libraries are ready to be downloaded.

On the client side, the Lib Manager helps build an execution environment for tailored apps. When an end user downloads an app, the Market Proxy handles the request and provides the tailored app instead of the original one. At the same time, the Lib Manager automatically checks local environment and downloads necessary third-party libraries. The downloaded libraries are individually stored on the device, and tailored apps then invoke these libraries through a dynamic loading mechanism.

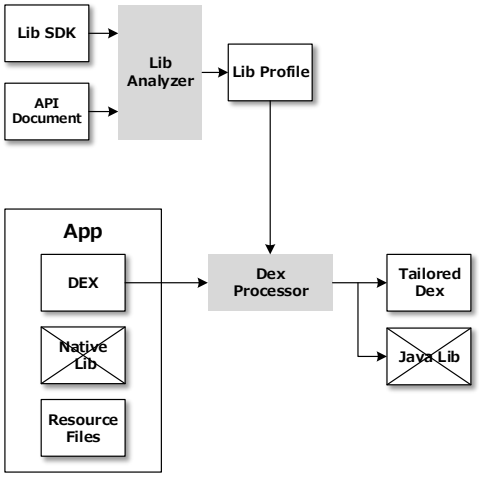

Fig. 5. App trimming: both native and java libraries are identified and separated from the original app.

APPCOMMUNE reduces the consuming of both network traffic and local storage through this sharing mechanism. In this way, the same library file will not be downloaded repeatedly. And through a centralized management of third-party libraries, APPCOMMUNE could immediately update the library and avoid the security threats due to the use of outdated libraries.

B. App Trimming

To achieve a centralized management of third-party libraries, we need to separate the integrated third-party libraries out from app. Figure 5 depicts the process of app trimming. There are two types of third-party libraries: java library (`.jar`

or `.aar`[3]) and native library (`.so`). We can directly extract native libraries because they are individually stored in an `apk` file. For each java library, we generate a library profile. According to this library profile, we identify the library code from the `dex` file then cut it off. At last, we find and adjust all library API invocation points in app code and generate a tailored `dex` file.

1) Library Profiling: We generate the profile of library by utilizing the open-source project LibScout [12], which selects class hierarchy as the base feature to generate the profile which is capable of identifying library and even pinpointing the exact library version. Considering that LibScout is originally designed for library identification, we further extend its implementation to make it suitable for app trimming and further app rewriting. In details, we add API statistics as another feature into the profile which helps us to locate the used APIs in app code. The API statistics include each API's parameter type, return value type, and the class hierarchy it belongs to. It contains no detailed class or method names (all user-defined names are marked as unified symbol), so it can still resist most code obfuscations.

2) Library Separating: By matching the `dex` file with all library profiles generated before, we can determine the detailed integrated libraries in the target app. Then, we decompile the `dex` file into `smali` [13] files, and resect the library code according to the class hierarchy of matched profile.

3) Invocation Points Patching: Finally, we utilize the API statistics in each matched library profile to locate library API invocation points in remained `smali` code (app code). We will modify these invocation points later in app rewriting (Section III-D).

C. Library Updating and Version Adapting

After app trimming, we abandon all integrated third-party libraries. Then we need to prepare the libraries with proper versions to match the trimmed app. As mentioned in Section II-C (insight c), we adopt a conservative updating strategy. On the premise of stability, we try to update them to latest versions as far as possible.

- For each detected library in a target app, if a new version is available, we first need to determine which APIs of this library are actually used in app code. We have obtained this information while app trimming as mentioned in Section III-B.
- Then we need to compare the new APIs provided by new version library and the ones used in app code. According to the detailed change types of APIs, we determine the compatibility between app and the new version library.
- If a major change exists in one of the used APIs, the update will not be accepted. Only when all used APIs have the same signatures with the ones provided by new library, we regard the new library compatible. Here we compare the new library with app code instead of old library, because only actually used APIs impact the compatibility. Even if there is a major change between two versions of library, it's still updatable as long as the major change isn't used by app code.
- Normally a minor change or patch change will not affect the compatibility. Considering special cases, we maintain a database for each API change on the server side. If any minor change or patch change causes any exception in subsequent tests, it will be recorded as a major change and will not be updated afterwards. For the library that cannot be updated to the latest version, we use a relatively new and suitable version instead and notify the app developer to update its code as soon as possible.

[3] `.aar` combines `.jar` and resource files. In our work, we don't deal with any resource file, and we would transform `.aar` into `.jar` directly. For convenience, we collectively call `.aar` and `.jar` as java library.

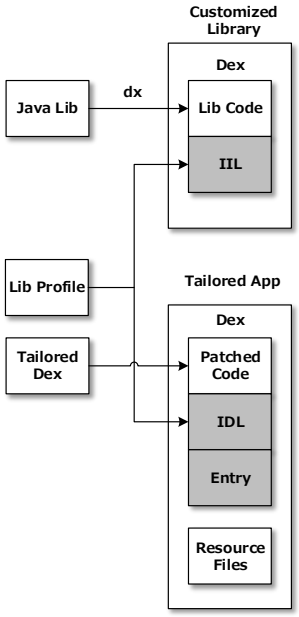

Fig. 6. App and library rewriting: dex files of both apps and libraries should be rewritten to add extra patching code.

D. App and Library Rewriting

To help tailored app access these separated matched libraries, we need rewrite both the app and these libraries.

To make external native library accessible, we rewrite the app code related to native library loading. Android system provides two system APIs for developers to load a native library: `System.loadLibrary` can only load a library from app's sandbox or system library path, while `System.load` can load any library according to the path parameter. Thus we replace the `System.loadLibrary` with `System.load` in app code, and change the parameter of `System.load` into the corresponding native library in Lib Manager.

For java library, our strategy is to replace the direct invoking to dynamic indirect invoking. As Figure 6 shows, we first use android platform tool `dx` to transform java library into `dex` code. Then, we add three pieces of extra code: *Interface*

Implementation Layer (IIL) in customized library, *Interface Declaration Layer* (IDL), and *Entry* in tailored app. According to the API statistics in library profile, we re-encapsulate all library's APIs into wrapper APIs which are declared in IDL and implemented in IIL. *Entry*, which will be invoked just when app starts, is responsible for loading the customized library and initializing an IIL instance. This instance is used to invoke the wrapper APIs directly. In the next, we patch all invocation points to invoke wrapper APIs. At last, we combine IIL and library code into a customized library `dex` file. And we combine IDL, *Entry* and the patched `dex` file into an entire `dex` file, which is repacked into tailored app with the original resource files.

We give a typical example to help illustrate our patching work: As Figure 7 shows, we re-encapsulate `sdkInitialize` (library API) into `com_facebook_Facebook_sdkInitialize` (wrapper API), which is declared in IDL and implemented in IIL. In *Entry*, the library `dex` is loaded (line 02) and an instance `idl` is initialized (line 04). Notice that `idl` is an instance of IIL (a part of library), but is casted into IDL (a part of app code) type. Therefore the wrapper APIs, which are implemented in library code and declared in app code, can be invoked directly through this instance. In this way, we centralize the java reflection options in *Entry* and reduce the changes made in each invocation point. As Figure 7 shows, the patched invocation point remains one line code as the original code.

We summarize the patching details for seven types of APIs as below:

- **static method:** Static method can be invoked directly without any context limitation, therefore it is a common kind of API. We directly invoke a static method typed API in the wrapper API. And the wrapper API has the same parameters and return value as the ones of original API.
- **public method**: Compared to static method, a normal public method's invocation relies on a context. For normal public methods, we add an extra parameter into the wrapper API. The invocation context should be passed through this extra parameter.
  Besides methods, we also need to deal with library's objects used in app code.
- **public class**: Library code may provide a public class for app developer to use directly. Library's public classes would be dynamically loaded, so they can't be used directly in app code. If app code declares variables in type of these public classes, we would switch them into `Object` type. Specially, if the parameters of wrapper APIs are in type of these public classes, they would be changed into `Object` as well. These `Object` typed parameters would be changed back and passed into original APIs in IIL.
- **public field**: Some library classes would provide public fields for app developer to access. For each public field, we add two extra public APIs responsible for reading and writing (`get()` and `set()`).
  The following cases are not suitable for dynamic loading and java reflection. We need to treat these APIs as part of app code.
- **public interface**: Interface typed class is also a common type of API (An example is given in Figure 2). It's a kind of callback that app code implements the interface declared by library code. Since some classes in app code are implemented from it, this interface typed class must be initialized before these classes in app code. Therefore, it doesn't support dynamic loading. We solve this challenge by adding library's interface typed class into IDL. Our IDL is also an interface typed class. We separate all library's interface typed classes out and add them into IDL in patched app. By this way, these APIs would be loaded together with classes of app code. And the remained part of library can be dynamically loaded normally.
- **abstract class**: An abstract class in library code provides abstract methods for app code to overload. Like interface typed class, we add abstract classes into IDL in the same way. Afterwards as a part of app code, the code in abstract class would be treated as normal app code. It means that the library's methods invoked in an abstract class would also be treated as library APIs.
- **extra component**: Some libraries also provide extra Android components (*Activity*, *Broadcast Receiver*, *Service*, and *Provider*) and require app developer to declare these components in `manifest.xml`. These components, which have been registered in system, can't be dynamically loaded either. We also separate these component classes out from library code and add them into app code.

E. Library Management

1) Downloading: On local device side, a terminal app, Lib Manager, is pre-installed and is responsible for downloading and managing customized libraries. APPCOMMUNE follows the basic mechanism of Android system and Lib Manager doesn't request root privilege. At first, the tailored app is downloaded and installed normally as ordinary ones. Simultaneously, Lib Manager judges whether all needed versions of libraries have been downloaded before. If lack of any library, Lib Manager would request it from Market Proxy and store it into the library store on the device (By default the customized libraries are stored in path `/data/local/tmp/`). Normally there would be only the latest version downloaded on the device for each library. In special cases, some older versions would be downloaded if some apps are not suiable for latest versions. The same library of same version would not be downloaded repeatedly. Specially, for a native library, there would be several versions for different platforms (e.g., arm, x86, etc.). Lib Manager can select the suitable version according to the device platform.

2) Execution: All customized libraries are set to be readable in the whole system. When the tailored app starts to run, the code of *Entry* will be executed first. *Entry* dynamically

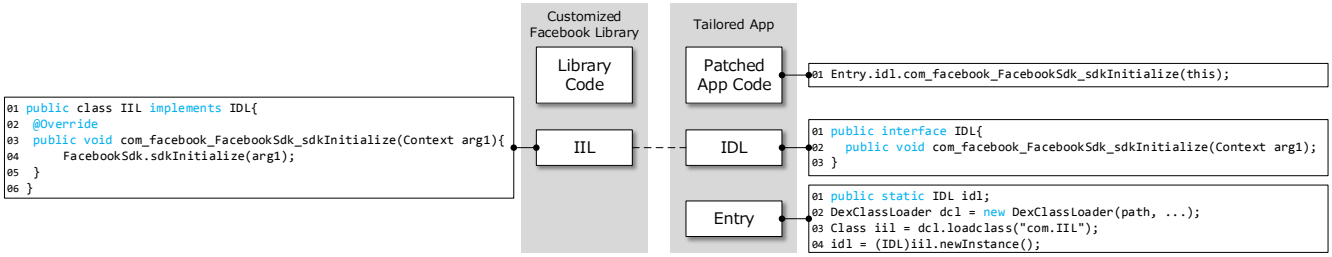

Fig. 7. A detailed example of our code patching work

loads the corresponding customized java library (`dex` format) and initializes the code for further invocations. The native library is directly accessed by `System.load` with their absolute file path. The tailored app only requests the library names, the detailed versions are decided by Lib Manager according to our library version adapting results on Market Proxy.

3) Further Updating: The further updates of tailored apps and customized libraries are also independent. When a new version of app is available, Lib Manager will notice the user to download. If a new version of library is available, Market Proxy will first determine whether it's compatible with each app. According to this matching result, all suitable apps are forced to use the new library. The developers of unsuitable apps would be notified to update their apps as soon as possible. Our Market Proxy supports Android smart update [14] to make the update more lightweight.

IV. Evaluation

A. Experimental Environment and Dataset

We deploy AppCommune on a workstation (server side) and 10 different Android devices (client side) to evaluate its usability. The workstation is a ThinkCentre with Core i7 @ 3.40GHz and 16GB of RAM running Ubuntu 16.04.

The Android devices include Nexus 4, 5, 6, 6P, Samsung S7, S8, Huawei Honor8, P9, Moto Z, Sony Xperia L and cover different Android OS from 4.4 to 7.0.

We establish two datasets to help conduct a comprehensive evaluation against AppCommune:

- **Lib Dataset**: To generate library profiles for library detection, we collect 212 distinct popular third-party libraries with 3,774 different versions. We download the latest version of library SDKs along with their historical versions (if available) from their official website or github repositories.
- **App Dataset**: In order to simulate the actual situation as far as possible, we invite 10 volunteers to provide their actual app lists (app name and version) in their mobile phones. According to these app lists, we download the corresponding apps from Google Play [15] and Tencent Android Market [16]. Totally we collect 502 apps and establish 10 app groups to simulate the mobile environment of each volunteer. The number of each app group is between 34 and 83.

B. Library Detection

TABLE I
Result of Library Detection

Total Apps	502
Total Detected Library Instances	2,695
Average Libraries in Each App	5.37
Distinct Kinds of Detected Libraries	73
Different Versions of Detected Libraries	623

TABLE II
Top 10 Detected Third-Party Libraries

Library	Number of App	Ratio
Gson [17]	225	44.8%
universal-image-loader [18]	99	19.7%
Nine Old Androids [19]	78	15.5%
Apache [20]	69	13.7%
Facebook [21]	63	12.5%
WeChat [22]	58	11.6%
Fresco [23]	57	11.4%
Crashlytics [24]	54	10.8%
OkHttp [25]	51	10.1%
Admob [26]	46	9.2%

We first utilize our extended version of LibScout to generate library profile for each library SDK. Then, we use the generated library profiles to identify the integrated libraries in our collected apps. As Table I shows, from all 502 collected apps, we totally detect 2,695 third-party library instances that cover 73 distinct libraries with 623 different versions. Table II summarizes the top 10 detected third-party libraries in our dataset. The most frequently used third-party library is Gson, which is contained by 44.8% collected apps and leads the ranking list. Six of ten in the list are commonly used utility libraries (e.g., Gson, universal-image-loader, etc.). It also includes social media library (like Facebook, Wechat), analytics library (like Crashlytics), and so on.

C. App Trimming and Rewriting

For each app, we first cut off its detected third-party libraries. Then we select the matched libraries from our lib dataset, try to replace the old version with a new version according to the strategy in Section III-C. In the end, we patch them into tailored apps and customized libraries. For each app,

TABLE III
DETAILS OF CODE PATCHING

	AVERAGE NUMBER IN EACH APP	RATIO
INVOKED LIBRARY APIS	541.7	NA
CHANGED CODE LINES	4,350.18	0.41%
ADDED CODE LINES	2,473.65	0.23%

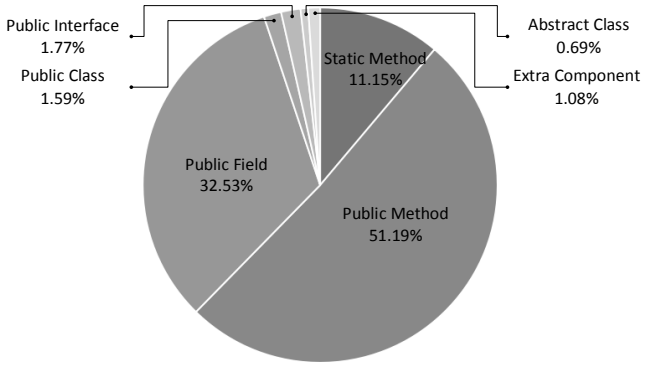

Fig. 8. The figure shows the percentage of all invoked API types.

we patch each library API invocation point and add some extra code (IDL and *Entry*).

Table III shows that, we find 541.7 different library APIs in each app on average. In response, for each app 4,350.18 lines of `smali` code are changed and 2,473.65 lines are added. Considering that a normal app often contains more than a million lines of `smali` code, the amount of our patched code is acceptable. The detailed distribution rate of invoked API type is shown in Figure 8. More than half (51.19%) of the APIs are in type of public methods.

In our experiment, we find most detected third-party libraries are able to be updated (to a new version, may not be the latest one). 231 of 2,695 (8.6%) detected library instances are not used by app actually (App developer adds them into app but not invokes any of their APIs). For these "dead" libraries, we directly remove them without a further code patching. For the remaining 2,464 libraries, we will check whether they can be updated to any new version and the app can keep stable as mentioned in Section III-C. As Figure 9 shows, 2,129 of 2,464 (86.4%) libraries can be updated at least one version. Specially, 1099 (44.6%) libraries can be updated to the latest versions. For example, the most common used library Gson, which is used by 44.8% apps has a 100% updated rate. Because it hasn't changed its APIs from its version 2.0 to 2.8.1. Finally, we generate only 206 versions of customized libraries to replace the 2,695 detected library instances (which cover 623 original versions of libraries).

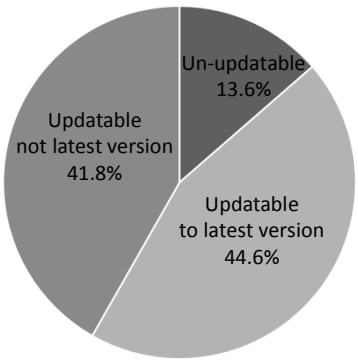

Fig. 9. The figure shows the library updatability in our experiment.

D. App Running and Stability

We prepare ten Android devices installed with Lib Manager to serve as client sides. According to each volunteer's original app list, every device downloads and installs the corresponding tailored apps to simulate the actual situation. Then, we evaluate tailored apps' stability in two steps.

In the first step, we manually run each app for five minutes and try our best to trigger more app code. Since most apps need to log in at first, so we select manual operation instead of automated execution. To trigger more patched APIs, we would register and login accounts, click all buttons, and we use adb (Android Debug Bridge) tool to trigger all extra components added by third-party libraries. We also consider the functions of detected libraries while performing operations. For example, we make more sharing-related operations for facebook-share library, and make more payment-related operations for paypal library. Here we extra add some code for exception handling and logging code into each API invocation point to help us obtain more details. After the execution, we check the logs to determine whether each API is triggered or any exception has occurred.

As a resutl, all 502 tailored apps run normally, no crash or exception occurs. Table IV shows the details about triggered APIs and the trigger ratio. Totally, we can trigger 63% of all patched APIs. Among all API types, public class and extra component have higher trigger ratios (93% and 89.8%). In most cases, public class contains the most commonly used functions so it's easy to be triggered. For extra component, we will trigger it forwardly as mentioned above. Although we still miss 37% of all patched APIs, the missed ones have the similar code formats as the triggered ones, so we can believe their effectiveness. On the other hand, in average, each app has 10,684.29 API trigger logs within five minutes. This value is extremely high and draws our attention. After checking the logs, we find some libraries would provide special APIs which are invoked continuously with a high frequency. We summarize the top five frequently invoked APIs in Table V.

In the second step, we request each volunteer to use our test device instead of their own mobile for 24 hours. In the meanwhile, volunteers just use each app as usual and notice whether any app is abnormal. In the end, they feedback their experiences. As a result, all tailored apps work normally and no functional abnormalities have occurred. This further proves

TABLE IV
API INVOCATIONS FOR EACH APP IN FIVE MINUTES APP RUNNING TEST

API TYPE	PUBLIC METHOD	STATIC METHOD	PUBLIC FIELD	PUBLIC CLASS	PUBLIC INTERFACE	ABSTRACT CLASS	EXTRA COMPONENT	TOTAL
INVOCATION LOGS	5,838.3	563.3	17.5	3,919.8	255.09	85.0	5.3	10,684.29
TRIGGERED APIS	185.3	36.2	98.5	8.0	5.8	2.2	5.3	341.3
PATCHED APIS	277.3	60.4	176.2	8.6	9.6	3.7	5.9	541.7
TRIGGER RATIO	66.8%	59.9%	55.9%	93%	60.4%	59.5%	89.8%	63.0%

TABLE V
API WITH HIGH FREQUENCY

LIBRARY	API	INVOCATION TIMES/MINUTE
Google Play Services	`GoogleApiClient;→connect`	8,384.8
Okio	`BufferedSink;→writeUtf8`	5,453.2
Okio	`BufferedSource;→read`	5,311.4
Dagger	`Linker;→requestBinding`	3,365.0
Google Play Services	`GoogleApiClient;→isConnected`	3,266.0

that our conservative strategy for library updating can ensure the app's stability.

E. Effectiveness

TABLE VI
AVERAGE FILE SIZE CHANGES BEFORE → AFTER APPCOMMUNE

DEX	6.48 MB→1.67MB (25.8%)
JAVA LIB	456.2 KB→461.8 KB (101.2%)
APK	14.3 MB→12.2 MB (85.3%)

TABLE VII
SPACE SAVING FOR 10 EXPERIMENTAL GROUPS

GROUP	APP	SPACE CHANGE	RATIO
1	52	743.6MB→664.3MB	89.3%
2	83	1,173.2MB→1,033.6MB	88.1%
3	34	496.2MB→440.1MB	88.7%
4	62	875.6MB→787.2MB	89.9%
5	38	534.3MB→465.9MB	87.2%
6	72	1,029.6MB→923.8MB	89.7%
7	40	597.4MB→532.3MB	89.1%
8	59	874.3MB→781.6MB	89.4%
9	47	627.1MB→548.7MB	87.5%
10	36	504.8MB→448.3MB	88.8%
TOTAL	523	7.46GB→6.63GB	88.9%

1) Space Saving: By managing the third-party libraries centrally, APPCOMMUNE can effectively reduce the size of app to save the network flow and device storage. We calculate the average file size changes of our samples, the result is shown in Table VI. On average, by separating all third-party libraries, the tailored `dex` file is only 25.8% of the original `dex` file in size (with a difference of 4.81 MB). And the tailored app is 85.3% of the original app in size (with a difference of 2.1MB). Compared with `dex` file, the difference reduces in app because the `dex` file would be compressed in an `apk` file. Due to the extra code IIL added into the library, the customized library is 101.2% of the original one in size.

For ten experimental groups, we calculate the whole occupied space: tailored apps, customized libraries and the Lib Manager `apk` file (which is only 0.98MB). Then we compare with the total size of corresponding original apps. The result is shown in Table VII. Totally, for all 523 apps (502 different apps and 21 popular ones which exist in multiple groups), APPCOMMUNE can save totally 830MB device space. The space saving ratios for all groups vary from 87.2% to 89.9%. In average, 11.1% device storage and network flow is saved for a normal mobile.

2) Security Promotion: To verify whether the library updating of APPCOMMUNE enhances the security of app, we select five publicly revealed security vulnerabilities to evaluate it. Table VIII gives the details about the vulnerabilities. The vulnerability of Dropbox is caused by an exposed *Activity* `com.dropbox.client2.android.AuthActivity`. Similarly, Mopub and Umeng also suffer from vulnerable components. Facebook has a vulnerability in its login process and Okhttp has a vulnerability that allows an attacker to bypass certificate pinning. According to the vulnerable library versions, we totally find 31 influenced apps. We utilize 360appscan [27], an on-line app vulnerability scanning service, to help detect the potential vulnerabilities.

As a result, the vulnerabilities of Dropbox, Mopub, and Umeng can be detected. According to the details of the other two vulnerabilities, we manually check the other apps and confirm that all selected apps are vulnerable. Then we use 360appscan or manual analysis to review the corresponding tailored apps. As expected, APPCOMMUNE has replaced all vulnerable libraries and all these tailored apps are safe. Therefore, APPCOMMUNE is effective in enhancing app security.

F. Performance

The workflow of APPCOMMUNE can be divided into two parts: static process on server side and dynamic execution on client side. Complex and time-consuming code patching process is performed on Market Proxy, therefore APPCOMMUNE doesn't bring much performance loss to app executions. Totally, we consume 9.5 hours to process 502 apps. Table IX shows the average time for each step of our static process on server side. In practice, the process of libraries (generating library profile and customized library) is pre-prepared. The average time to prepare a new uploaded app for downloading is only 68.1 seconds (library detection and generating tailored

TABLE VIII
THE VULNERABILITY DETECTION RESULTS BEFORE AND AFTER THE LIBRARY UPDATING

LIBRARY	INFL VERSIONS	CVE	INFL APPS	DETECTING ENGINE	BEFORE→ AFTER UPDATING
Dropbox	1.5.4-1.6.1	CVE-2014-8889	4	360appscan	4→0
Facebook	3.15	-	5	manual check	5→0
Mopub	3.10-4.3	-	11	360appscan	11→0
Okhttp	3.0.0-3.1.1	CVE-2016-2402	7	manual check	7→0
Umeng	3.0.0-3.1.3	-	4	360appscan	4→0

TABLE IX
CONSUMED TIME IN STATIC PROCESS

	AVERAGE TIME
LIB PROFILE	5.129 sec/lib
PROCESS CUSTOMIZED LIB	5.782 sec/lib
LIB DETECTION	33.997 sec/app
PROCESS TAILORED APP	34.268 sec/app

app), which is acceptable. On the cient side, the tailored app still follows the execution principle of Android system, and runs as normal apps. Although the process of customized library loading would consume extra time, the app user will not feel any obvious lags. On average, each app extra consumes 0.5 second on each start-up process. APPCOMMUNE only brings ignorable performance loss while the execution of apps.

V. DISCUSSION

APPCOMMUNE makes third-party library sharing available and proves that automatic updates of third-party libraries are feasible in theory. However, some limitations still exist in practice. For example, the app developer doesn't agree with the automatic update or some developers modify the original libraries personally which causes incompatibilities after automatic updates. Fortunately these are rare cases. It's more important to raise app developers' awareness of safety. APPCOMMUNE's app trimming is based on static analysis, which means it would suffer from app code protections. To address this challenge, we can utilize AppSpear [28], an open-source unpacking system which can transform a packed app into unpacked one automatically.

VI. RELATED WORK

Various studies emphasize the security problems brought by third-party libraries. Researches [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] showed that third-party library would leak privacy, abuse permissions, and even bring security vulnerabilities. Erik *et al.* [1] proposed the library outdateness problem in Android and even explored the reasons. We verify their findings in our experiments and we firstly propose the solution for the library outdateness problem.

These security problems have motivated approaches to detect third-party libraries. Approaches [31] [30] [39] were early proposed which select package name as a simple feature. Some other works [40] [41] [42] are based on machine learning or code clustering, which extract features from app code. LibScout [3] use class hierarchy as feature which is extracted from original library. Approaches have also been proposed to solve these security problems. Some approaches select to limit third-party libraries' permissions and behaviors. AdSplit [4] puts ad library code into separated processes. AdDroid [5] proposes a new Android framework to allow ad library's privilege separation. PEDAL [6] limits ad library's behaviors by inline reference monitoring. FlexDroid [7] enforces in-app privilege separation. And CompARTist [43] partitions apps directly at compile-time into isolated, privilege-separated compartments for the host app and the included third-party libraries. APKLancert [8] radically removes the malware and ad libraries from the repacked apps. These approaches are more suitable for malicious libraries, because the separating or removing would impact the libraries' functions. Comparing with them, APPCOMMUNE reduces the impact of the vulnerability by automatically updating the third-party libraries which doesn't impact the behaviors of libraries.

VII. CONCLUSION

This paper proposes APPCOMMUNE, an automated system to achieve a third-party library sharing strategy. APPCOMMUNE separates all third-party libraries from apps on the server side, and centrally manages the libraries on client side. Multiple apps would share one library which can avoid duplicated libraries and save the bandwidth as well as storage. The managed libraries are automatically updated, which reduces the threats of vulnerabilities in outdated third-party libraries. Our experiment demonstrates that APPCOMMUNE is a feasible management and it can effectively help update the third-party libraries and reduce the file size of apps.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments which greatly help to improve the manuscript. This work is partially supported by the Key Program of National Natural Science Foundation of China (Grant No.U1636217), the General Program of National Natural Science Foundation of China (Grant No.:61872237), the National Key Research and Development Program of China (Grant No.2016YFB0801200), the Major Project of Ministry of Industry and Information Technology of China (Grant No.[2018] 282). We especially thank the Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial joint Institute of FinTech Security*. This work is funded in part by Nanjing Turing Artificial Intelligence Institute.

REFERENCES

[1] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android," in *CCS*, 2017.

[2] M. Chi, "LibDetector: Version Identification of Libraries in Android Applications," in *Rochester Institute of Technology*, 2016.

[3] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and its Security Applications," in *CCS*, 2016.

[4] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications." in *USENIX*, 2012.

[5] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege Separation for Applications and Advertisers in Android," in *ASIACCS*.

[6] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps," in *MobiSys*, 2015.

[7] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: Enforcing In-App Privilege Separation in Android." in *NDSS*, 2016.

[8] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu, "APKLancet: Tumor Payload Diagnosis and Purification for Android Applications," in *AsiaCCS*, 2014.

[9] "NoograNuts in Google Play," https://play.google.com/store/apps/details?id=com.bengigi.noogranuts.

[10] "MAPSME in Google Play," https://play.google.com/store/apps/details?id=com.mapswithme.maps.pro.

[11] "Facebook Changelog," https://developers.facebook.com/docs/android/change-log-4x, 2018.

[12] "LibScout Project," https://github.com/reddr/LibScout, 2017.

[13] "Smali and Baksmali," https://github.com/JesusFreke/smali.

[14] "Android Smart Update," https://techcrunch.com/2012/08/16/google-turns-on-smart-updates-for-android-apps/.

[15] "Google Play," https://play.google.com/store.

[16] "Tencent Android Market," http://sj.qq.com/myapp/.

[17] "Gson," https://github.com/google/gson.

[18] "Universal-Image-Loader," https://github.com/nostra13/Android-Universal-Image-Loader.

[19] "Nine Old Androids," https://github.com/JakeWharton/NineOldAndroids.

[20] "Apache HttpClient for Android," http://hc.apache.org/.

[21] "Facebook," https://developers.facebook.com.

[22] "WeChat," http://open.wechat.com.

[23] "Fresco," https://github.com/facebook/fresco.

[24] "Crashlytics," https://github.com/crashlytics/cannonball-android.

[25] "OkHttp," http://square.github.io/okhttp/.

[38] C. Zuo, Z. Lin, and Y. Zhang, "Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps."

[26] "Admob," https://www.google.com/admob/.

[27] "Appscan 360," http://appscan.360.cn/.

[28] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "AppSpear: Automating the Hidden-Code Extraction and Reassembling of Packed Android Malware," in *Journal of Systems and Software*, 2018.

[29] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating User Privacy in Android Ad Libraries," in *MoST*, 2012.

[30] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal Analysis of Android Ad Library Permissions," *arXiv preprint arXiv:1303.0857*, 2013.

[31] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile in-App Advertisements," in *WISEC*, 2012.

[32] S. Son, D. Kim, and V. Shmatikov, "What Mobile Ads Know About Mobile Users." in *NDSS*, 2016.

[33] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization." in *USENIX*, 2013.

[34] H. Wen, J. Li, Y. Zhang, and D. Gu, "An Empirical Study of SDK Credential Misuse in iOS Apps," in *APSEC*, 2018.

[35] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show Me The Money! Finding Flawed Implementations of Third-Party in-App Payment in Android Apps," in *NDSS*, 2017.

[36] X. Zhu, J. Li, Y. Zhou, and J. Ma, "AdCapsule: Practical Confinement of Advertisements in Android Applications," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[37] C. Zuo and Z. Lin, "Smartgen: Exposing Server Urls of Mobile Apps with Selective Symbolic Execution," in *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[39] K. Chen, P. Liu, and Y. Zhang, "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets," in *ICSE*, 2014.

[40] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection," in *ISSTA*, 2015.

[41] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps," in *ICSE*, 2016.

[42] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and Precise Third-Party Library Detection in Android Markets," in *ICSE*, 2017.

[43] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android," in *CCS*, 2017.